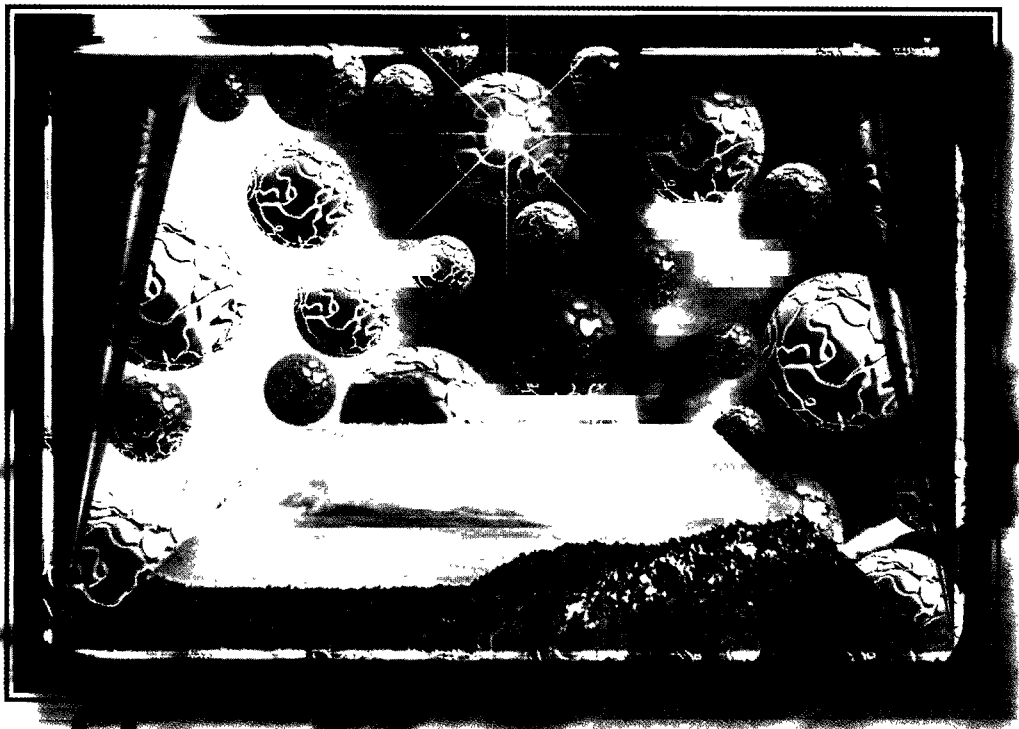
Системное программирование на C++ для Unix

• Передовые методы программирования на C++ • Создание приложений клиент/сервер
• Многочисленные примеры классов и программ



bhv

Теренс Чан

В этой книге, рассчитанной прежде всего на опытных программистов, содержится обзор всех современных технологий, используемых при создании сложных системных приложений для среды UNIX.

Системное программирование на C++ для UNIX

Особое внимание уделяется проблемам, с которыми сталкиваются разработчики при создании сетевых приложений и приложений клиент/сервер, баз данных, компиляторов, операционных систем и приложений САПР. Подробно описываются приемы создания программ на C++, отличающихся компактностью, легкостью в сопровождении и простотой переноса на большинство UNIX- и POSIX-совместимых систем (например, Windows NT).

В книгу включены примеры программ, в которых демонстрируются принципы создания классов и приложений с помощью стандартных функций и классов ANSI, POSIX и UNIX. Кроме того, в ней приведены исходные тексты готовых классов, предназначенных для решения задач межпроцессного взаимодействия на основе гнезд и TLI, многопотокового программирования и организации удаленного вызова процедур. Все они могут быть встроены во вновь создаваемые приложения, что позволит программисту сэкономить время и повысить качество программ.

Об авторе

Теренс Чан ведет курс "Современные средства создания программного обеспечения для UNIX" в университете г. Беркли (Калифорния) и его филиале в г. Санта-Круз. Он имеет более чем двенадцатилетний опыт работы в области создания системных программных продуктов для всех основных UNIX-платформ на языках C и C++.

bhv

Системное программирование на C++ для UNIX

UNIX System Programming Using C++

Terrence Chan

To join a Prentice Hall PTR internet mailing list:
point to http://www.prenhall.com

Prentice Hall PTR
Upper Saddle River, New Jersey 07458

http://www.prenhall.com.

Теренс Чан

Системное программирование на C++ для Unix

Перевод с английского С. Тимачева
под редакцией М. Коломыцева

BHV, Киев, 1997

УДК 681.3.06

Теренс Чан
Системное программирование на C++ для UNIX: Пер. с англ. — К.: Издательская группа BHV, 1997. — 592 с. ISBN 5-7315-0013-4

В книге содержится обзор современных технологий разработки сложных системных приложений для среды UNIX. Приведены многочисленные примеры программ, демонстрирующие принципы создания классов и приложений с помощью стандартных функций и классов ANSI, POSIX, UNIX; включены исходные тексты готовых классов, которые могут быть встроены во вновь создаваемые приложения, что позволит программисту сэкономить время и повысить качество своих программ. Особое внимание уделяется реальным проблемам, с которыми сталкиваются разработчики приложений клиент/сервер и других программных продуктов.

Предназначена в первую очередь для специалистов, желающих овладеть передовыми методами программирования на C++ для UNIX.

ISBN 0-13-331562-2
© Prentice Hall PTR, 1997

ISBN 5-7315-0013-4
© Издательская группа BHV, Киев, 1997
А.А. Стеценко, обложка

Предисловие

Содержание этой книги отражает мой опыт преподавания передовых методов программирования на языках C и C++ для ОС UNIX на курсах при университете штата Калифорния (в г. Беркли и г. Санта-Крузе). Целью курсов было научить студентов эффективным методам программирования на указанных языках с использованием системных вызовов UNIX. В частности, студенты изучали следующий материал:

- Передовые методы программирования на ANSI C и C++, в том числе применение указателей на функции и создание функций, принимающих переменное число аргументов.
- Библиотечные функции ANSI C и стандартные классы C++, возможность сокращения с их помощью времени на разработку и максимальное повышение переносимости приложений.
- Структура ядра и системные вызовы ОС UNIX. (Знание этого материала позволяет пользователям создавать эффективные программы, предназначенные для манипулирования системными ресурсами, например, файлами, процессами и системной информацией, а также новые операционные системы.)
- Методика создания сетевых многозадачных приложений клиент/сервер, работающих на разнородных UNIX-платформах.

Цель книги — ознакомить читателей со всеми этими методами и концепциями. Кроме того, по каждой теме здесь даются более подробные пояснения и примеры, нежели в рамках курсов. Читатели, таким образом, смогут лучше разобраться в рассматриваемом предмете и изучать его в удобном для себя темпе. В книге также описаны новейшие из используемых в ОС UNIX методы программирования, основанные на удаленных вызовах процедур

и многопоточных программах. Эти методы очень важны для разработки распределенных приложений клиент/сервер, предназначенных для симметричных мультипроцессорных систем и сетевых вычислительных сред.

Вся информация излагается с использованием языка C++. Это объясняется тем, что в последние несколько лет все больше и больше высококвалифицированных разработчиков программного обеспечения используют этот язык для создания приложений. Причина состоит в том, что язык C++ в отличие от других процедурных языков программирования обеспечивает гораздо более строгий контроль типов и содержит объектно-ориентированные программные конструкции. Указанные особенности очень полезны для разработки и сопровождения крупномасштабных и достаточно сложных приложений для ОС UNIX.

В книге описывается язык программирования C++, основанный на проекте стандарта ANSI/ISO C++ [1—3]. Этому проекту стандарта соответствует большинство новейших компиляторов C++, поставляемых различными фирмами-производителями компьютерных систем (например, Sun Microsystems Inc., Microsoft Corporation, Free Software Foundation и др.).

Помимо языка C++ в книге описаны некоторые важнейшие библиотечные функции языка C, определенные в стандарте ANSI C [4]. Стандартными классами C++ и интерфейсами прикладного программирования ОС UNIX они не охвачены, поэтому пользователям важно иметь о них представление, с тем чтобы расширить базу своих знаний и возможности выбора таких функций.

В книге рассматриваются следующие разновидности операционной системы UNIX: UNIX System V.3, UNIX System V.4, BSD UNIX 4.3 и 4.4, Sun OS 4.1.3 и Solaris 2.4. Последние две системы принадлежат фирме Sun Microsystems. Система Sun OS 4.1.3 построена на основе BSD 4.3 с расширениями UNIX System V.3, а Solaris 2.4 — на базе UNIX System V.4.

Несмотря на то что основной акцент в книге сделан на программировании для UNIX-систем, подробно рассматриваются и стандарты POSIX.1, POSIX.1b и POSIX.1c Института инженеров по электротехнике и радиоэлектронике (Institute of Electrical and Electronics Engineering). Цель — помочь системным программистам в разработке приложений, которые можно будет свободно переносить на разные UNIX-системы, а также на POSIX-совместимые системы (например, VMS и Windows NT). Это весьма важно, поскольку большинство коммерческих программных продуктов должно работать на различных платформах. Следовательно, стандарты POSIX и ANSI могут помочь пользователям в создании приложений с высокой степенью независимости от платформы.

Предполагаемый круг читателей

Книга рассчитана на опытных разработчиков ПО и администраторов систем, которые трудятся над созданием сложных системных приложений в среде UNIX. Разрабатываемые ими продукты могут включать сетевые

приложения клиент/сервер, распределенные базы данных, операционные системы, компиляторы, средства автоматизированного проектирования и т.д.

Читатели должны быть знакомы с языком C++, основанным на версии AT&T 3.0 (или более поздней), и иметь опыт самостоятельной разработки прикладных программ на C++. Кроме того, они должны быть знакомы хотя бы с одной версией ОС UNIX (например, UNIX System V). В частности, необходимо знать архитектуру файловой системы UNIX, методику создания пользовательских бюджетов и управления ими, методы управления доступом к файлам, методы управления заданиями. Тем читателям, которым нужно лишь освежить в памяти знания об ОС UNIX, рекомендуем обратиться к любому пособию начального уровня по этой системе.

Содержание книги

Хотя в книге широко освещаются язык программирования ANSI C++ и API ОС UNIX, основной акцент делается на описании библиотечных функций C и, в первую очередь, на доведении до читателей следующей информации:

- назначение функций;
- соответствие функций стандарту (стандартам);
- методика применения функций;
- примеры их использования;
- реализация функций в UNIX-системе (по необходимости);
- особые замечания по применению (например, в случае противоречия между стандартами UNIX и POSIX).

Автор ни в коем случае не собирается делать из этой книги справочник системного программиста UNIX. Так, здесь описываются прототипы функций и файлы заголовков, необходимые для работы с библиотечными функциями ANSI и функциями API ОС UNIX, но ни коды ошибок, которые могут возвращать эти функции, ни архивы, ни разделяемые библиотеки, необходимые пользовательским программам, подробно не рассматриваются. Эту информацию читатель может найти либо на man-страницах, посвященных соответствующим функциям, либо в руководствах программиста, прилагаемых к компьютерным системам.

Книга построена следующим образом:

- В главе 1 излагается история создания языка программирования C++ и различных версий ОС UNIX. Описываются также стандарты ANSI/ISO C, ANSI/ISO C++, IEEE POSIX.1, IEEE POSIX.1b и IEEE POSIX.1c.
- Главы 2 и 3 содержат обзор проекта стандарта ANSI/ISO C++ и методов объектно-ориентированного программирования. Подробно описываются классы потоков ввода-вывода C++, шаблоны функций, методы обработки исключительных ситуаций.

- В главе 4 описываются библиотечные функции ANSI C.
- В главе 5 дается обзор интерфейсов прикладного программирования (API) UNIX и POSIX. Излагается информация о специальных файлах заголовков и опциях, определяющих режимы компиляции, в соответствии с различными стандартами.
- В главах 6 и 7 перечислены API, предназначенные для работы с файлами ОС UNIX и стандарта POSIX.1. Эти API служат для управления файлами различных типов. Описаны также методы блокировки файлов, используемые для синхронизации файлов в мультипроцессорной среде.
- В главе 8 изложены методы создания процессов и управления процессами в UNIX и POSIX.1. Прочитав эту главу, читатели смогут писать собственные приложения, такие как shell в UNIX.
- В главе 9 изложены методы обработки сигналов в UNIX и POSIX.1;
- Глава 10 содержит сведения о методах межпроцессного взаимодействия в UNIX и POSIX.1b, выполняющих важную роль в создании распределенных приложений клиент/сервер.
- В главе 11 освещены передовые методы сетевого программирования с помощью гнезд (socket) UNIX и интерфейса транспортного уровня (TLI).
- В главе 12 описаны удаленные вызовы процедур. Эта информация представляет интерес при разработке независимых от сетевого транспортного протокола приложений клиент/сервер, функционирующих в среде разнородных UNIX-платформ.
- В главе 13 изложены методы многопоточного программирования, которые позволяют приложениям эффективно использовать ресурсы тех мультипроцессорных систем, в которых они выполняются.

Отметим, что хотя в этой книге используется C++, основное внимание уделяется *вовсе не* методам объектно-ориентированного программирования, характерным для этого языка. Причина состоит в том, что некоторые читатели, скорее всего, еще не знакомы с системным программированием для UNIX и (или) языком C++, поэтому им трудно будет одновременно изучать методы объектно-ориентированного и системного программирования. Тем не менее в книге приведено описание многих полезных классов C++, используемых для организации межпроцессного взаимодействия, гнезд, интерфейса транспортного уровня (TLI), удаленных вызовов процедур и многопоточного программирования. Эти классы скрывают детали низкоуровневого интерфейса программирования применительно к перечисленным системным функциям, могут расширяться и встраиваться в пользовательские приложения, позволяя сократить время и затраты на их разработку.

Примеры программ

В книге приведено множество примеров программ, иллюстрирующих использование классов C++, библиотечных функций и системных API. Все

эти примеры компилировались компилятором C++ от Sun Microsystems (версия 4.0) и тестировались на рабочей станции Sun SPARC-20, работающей под управлением ОС Solaris 2.4. Кроме того, примеры компилировались и тестировались с помощью компилятора GNU g++ от Free Software Foundation (версия 2.6.3) на рабочей станции Sun SPARC-20. Поскольку компиляторы GNU g++ можно переносить на разные аппаратные платформы, представленные в книге программы-примеры должны нормально работать и на других платформах (например, под управлением HP-UX от Hewlett-Packard и AIX от IBM).

Для более тесного знакомства с данным предметом читателям рекомендуется проверить приведенные примеры на своих системах. Электронную копию этих программ можно загрузить по анонимному ftp с *ftp.prenhall.com* (tar-файл с примерами находится в каталоге */pub/ptr/professional_computer_science.w-022/chan/unixsys*). В этом архивном tar-файле есть файлы README с описаниями программ и перекрестными ссылками на главы книги. Наконец, читатели могут присылать свои сообщения автору по электронной почте (*twc@netcom.com*).

Выражения признательности

Я хотел бы поблагодарить Питера Коллинсона, Джеффа Гитлина, Чи Куонга, Фрэнка Митчелла и свою жену Джессику Чан за внимательное рецензирование рукописи. Большинство из их ценных замечаний отражено в окончательном варианте книги. Кроме того, я хотел бы выразить признательность Грегу Денчу, Нику Радхуберу и Брэту Бартоу за помощь в подготовке и издании книги.

Наконец, я благодарен своим бывшим студентам — слушателям курсов при университете штата Калифорния в Санта-Крузе, которым я преподавал предмет *Advanced UNIX System Calls*. Они давали мне ценнейшую информацию для размышлений и для доработки материала книги.

Теренс Чан

Литература

1. Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.
2. Andrew Koenig, *Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++ (Committees: WG21/NO414, X3J16/94-0025)*, January 1994.
3. Bjarne Stroustrup, *Standardizing C++. The C++ Report, Vol. 1, No. 1*, 1989.
4. American National Standard Institute, *American National Standard for Information Systems — Programming Language C, X3.159 — 1989*, 1989.

Об авторе

Теренс Чан ведет курсы по передовым методам программирования и новейшим средствам разработки программного обеспечения для ОС UNIX при университете штата Калифорния в Беркли и Санта-Крузе. За его плечами более чем двенадцатилетний практический опыт проектирования и разработки крупномасштабных программ на C++ и C для всех основных UNIX-платформ. В настоящее время Теренс Чан является консультантом по проектированию объектно-ориентированных, распределенных и многопоточных приложений архитектуры клиент/сервер.

ГЛАВА 1

Операционная система UNIX и стандарты ANSI

Операционная система UNIX была создана в конце 60-х годов, и с тех пор появилось множество различных ее версий для разных компьютерных систем. Последние версии системы UNIX развились из AT&T System V и BSD 4.x UNIX. Многие производители компьютеров добавляли к этим разновидностям UNIX собственные расширения, создавая таким образом новые версии UNIX. В конце 80-х годов AT&T и Sun Microsystems работали над совместным проектом по созданию UNIX System V release 4, который был, по сути, попыткой разработать стандарт на UNIX для компьютерной индустрии. Попытка эта оказалась не совсем успешной, поэтому UNIX System V.4 сегодня используют лишь немногие производители компьютеров.

Тем не менее в конце 80-х годов несколько организаций предложили ряд стандартов на UNIX-подобную операционную систему и среду программирования на языке C. Эти стандарты построены в основном на UNIX и не влекут за собой существенных изменений в системах, выпускаемых фирмами-производителями, что облегчает их практическое внедрение. Два из этих стандартов, ANSI C и POSIX (Portable Operating System Interface — интерфейс переносимых операционных систем), разработаны Американским национальным институтом стандартов (ANSI) и Институтом инженеров по электротехнике и радиоэлектронике (IEEE). Эти организации имеют очень большое влияние на разработку стандартов для данной отрасли — большинство производителей компьютеров сегодня предлагают UNIX-системы, соответствующие стандартам ANSI C и POSIX.1 (последний представляет собой подмножество стандартов POSIX).

Большинство этих стандартов определяют требования к операционной системе, которая должна эффективно выполнять приложения, написанные

на С. Приложения, соответствующие названным стандартам, должны легко переноситься на другие системы, отвечающие этим же стандартам. Это особенно важно для высококвалифицированных системных программистов, широко использующих функции интерфейсов прикладного программирования (API) системного уровня (которые включают библиотечные функции и системные вызовы). Дело в том, что не во всех UNIX-системах имеется стандартный набор системных API.

Более того, даже некоторые общие API могут быть в разных UNIX-системах реализованы по-разному (например, API *fcntl* в UNIX System V может использоваться для блокировки и освобождения файлов, а в BSD UNIX он такой способностью не обладает). Стандарты ANSI C и POSIX требуют, чтобы все соответствующие им системы содержали одинаковый набор стандартных библиотек и системных API; эти стандарты определяют также сигнатуры (тип данных, число аргументов, возвращаемое значение) и порядок работы этих функций во всех системах. Таким способом добиваются того, что программы, которые используют эти функции, можно переносить на другие системы, соответствующие данным стандартам.

Большинство функций, определенных в этих стандартах, представляют собой подмножество функций, имеющихся в большинстве UNIX-систем. Комитеты ANSI C и POSIX все же предложили ряд функций собственной разработки, но эти функции лишь привносят дополнительные неоднозначность и неопределенность в некоторые существующие версии UNIX. Опытный разработчик, имеющий дело с UNIX и C, легко разберется в упомянутых стандартах. Поддержка этих стандартов производителями компьютеров — тоже несложная задача.

Цель данной книги — познакомить пользователей с новейшими методами программирования для UNIX-систем, в том числе научить их писать переносимые и легко сопровождаемые коды. Для обеспечения второго пункта сформулированной цели нужно ознакомить пользователей с функциями, определенными в различных стандартах, и с функциями, имеющимися в ОС UNIX. Лишь при этом условии пользователи смогут сделать разумный выбор в пользу тех или иных функций и API.

Далее в этой главе дается обзор стандарта ANSI C, проекта стандарта ANSI/ISO C++ и стандартов POSIX. В следующих главах более подробно описываются функции API, определенные в этих стандартах, а также функции API ОС UNIX.

1.1. Стандарт ANSI C

В 1989 году Американский национальный институт стандартов (ANSI) предложил стандарт языка программирования C (X3.158-1989), целью которого была стандартизация конструкций и библиотек этого языка. Названный стандарт, широко известный как *стандарт ANSI C*, стал попыткой унификации языка C, создания его реализации, поддерживаемой во всех компьютерных системах. Сегодня большинство поставщиков вычислительной

техники продолжают поддерживать стандартный набор конструкций и библиотек языка C, предложенный Б. Керниганом и Д. Ритчи (широко известен как *K&R C*), но пользователи могут за дополнительную плату инсталлировать инструментальный пакет программ ANSI C.

Ниже перечислены основные отличия ANSI C от K&R C:

- наличие прототипов функций;
- поддержка описателей типов данных *const* и *volatile*;
- поддержка локализации, т.е. разработки программного обеспечения, отвечающего потребностям различных стран и культур;
- возможность использовать указатели на функции без их разыменования.

Хотя основное внимание в этой книге уделяется методике программирования на C++, читатели все равно должны быть знакомы со стандартом ANSI C. Дело в том, что многие стандартные библиотечные функции C не охватываются стандартными классами C++, поэтому почти все программы на C++ вызывают одну или несколько стандартных библиотечных функций C (например, функцию *strlen* или функции, предназначенные для получения значения текущего времени суток). Кроме того, читатели, которые, возможно, уже начали переносить свои C-приложения на C++, найдут в данном разделе описание совпадающих конструкций этих двух языков и различий между ANSI C и C++. Это должно облегчить переход с языка ANSI C на C++.

В ANSI C принят используемый в C++ метод создания прототипов функций, согласно которому для определения и объявления функции необходимо указать ее имя, тип данных аргументов и тип возвращаемых значений. Прототипы функций позволяют компиляторам ANSI C проверять пользовательские программы на предмет наличия вызовов функций, которые получают неверное число аргументов или аргументы несовместимых типов. Таким образом устраняется наиболее серьезный недостаток компиляторов K&R C, заключающийся в том, что неверные вызовы функций в пользовательских программах на этапе компиляции часто не обнаруживаются, но при выполнении приводят к краху программ.

В следующем примере определяется функция *foo*, которая должна принимать два аргумента. Первый аргумент, *fmt*, имеет тип данных *char**, а второй — тип данных *double*. Функция *foo* возвращает значение *unsigned long*:

```
unsigned long foo (char* fmt, double data)
{
    /* тело функции foo */
 }
```

Для того чтобы объявить указанную функцию, пользователь просто берет приведенное выше определение, удаляет тело и заменяет его точкой с запятой:

```
unsigned long foo (char* fmt, double data);
```

Для функций, принимающих переменное число аргументов, в определениях и объявлениях в качестве последнего аргумента должно стоять многоточие:

```
int printf(const char* fmt, ...);

int printf(const char* fmt, ...)
{
    /* тело функции printf */
}
```

Ключевое слово *const* объявляет, что некоторые данные не подлежат изменению. Например, объявление приведенным выше прототипом функции аргумента *fmt* типа *const char** означает, что функция *printf* не может изменять данные в массиве символов, который передается ей как фактическое значение аргумента *fmt*.

Ключевое слово *volatile* объявляет, что значения некоторых переменных могут изменяться неизвестным для компилятора образом, "намекая" алгоритму оптимизации компилятора на то, что "избыточные" операторы с объектами *volatile* удалять нельзя. Например, в представленном ниже примере определяется переменная *io_Port*, которая содержит адрес порта ввода-вывода системы. Два оператора, следующие за определением, должны ожидать прибытия из порта ввода-вывода 2 байтов данных и сохранять только второй байт:

```
char get_io()
{
    volatile char* io_Port = 0x7777;
    char ch = *io_Port;          /* прочитать первый байт данных */
    ch = *io_Port;               /* прочитать второй байт данных */
}
```

Если переменную *io_Port* в этом примере не объявить как *volatile*, то при компиляции программы компилятор может удалить второй оператор *ch = *io_Port*, так как он считается лишним при наличии первого.

Описатели типов данных *const* и *volatile* имеются и в C++.

Язык ANSI C поддерживает национальные алфавиты, допуская использование в программах символов расширенной кодировки. Для хранения одного такого символа требуется больше 1 байта. Эти символы применяются в тех странах, где набор ASCII не является стандартом. Например, для представления каждого символа корейского алфавита требуется по 2 байта. Кроме того, в ANSI C есть функция *setlocale*, которая позволяет пользователям задавать формат представления дат, денежных единиц и вещественных чисел. Например, в большинстве стран дата выводится в формате число/месяц/год, а в США — в формате месяц/число/год.

Вот прототип функции *setlocale*:

```
#include <locale.h>

char setlocale ( int category, const char* locale );
```


Прототип функции *setlocale* и возможные значения аргумента *category* объявляются в файле заголовков <locale.h>. Значения *category* показывают, к какому типу (типам) выводимой информации будет применена новая локальная среда. Некоторые возможные значения этого аргумента приведены в таблице.

Значение аргумента *category*	Объект влияния
LC_CTYPE	Макросы, определяемые в файле <ctype.h>
LC_TIME	Формат даты и времени, возвращаемых функциями *strftime*
LC_NUMERIC	Форматы представления чисел функциями *printf* и *scanf*
LC_MONETARY	Формат представления денежной единицы, возвращаемой функцией *localeconv*
LC_ALL	Соответствует одновременному действию всех перечисленных значений

Возможное значение аргумента *locale* — это строка символов, которая определяет, какую локальную среду нужно использовать. Значения C, POSIX и en_US соответствуют локальным средам UNIX, POSIX и US. По умолчанию все процессы в системе, на которую распространяется стандарт ANSI C или POSIX, выполняют во время запуска вызов, эквивалентный следующему:

```
setlocale( LC_ALL, "C" );
```

Таким образом, все запускаемые процессы имеют известную локальную среду. Если аргумент *locale* имеет значение NULL, то функция *setlocale* возвращает текущее значение *locale* вызывающего процесса. Если аргумент *locale* имеет значение "" (пустая строка), то функция *setlocale* ищет значение этого аргумента в переменной среды LC_ALL, в переменной среды с именем, совпадающим со значением аргумента *category*, и, наконец, в переменной среды LANG (именно в таком порядке).

Функция *setlocale* определена в стандарте ANSI C, а также соответствует стандарту POSIX.1.

Согласно ANSI C, указатель на функцию можно использовать как имя функции. При вызове функции, адрес которой содержится в этом указателе, разыменования не требуется. Например, следующими операторами определяется указатель на функцию *funcptr*, который содержит адрес функции *foo*:

```
extern void foo (double xyz, const int* lptr);
void (*funcptr)(double, const int*) = foo;
```

Функция *foo* может вызываться либо путем прямого вызова *foo*, либо посредством *funcptr*. Следующие два оператора функционально эквивалентны:

```
foo (12.78, "Hello world");
funcptr (12.78, "Hello world");
```

В K&R C для вызова *foo* необходимо разыменовать *funcptr*. В синтаксисе K&R C оператором, эквивалентным приведенному выше, будет

```
(*funcptr)(12.78, "Hello world");
```

В C++ поддерживаются оба эти варианта использования указателей на функции.

В ANSI C определен также набор символов *cpp* (препроцессора C), которые допускается применять в пользовательских программах. Фактические значения присваиваются этим символам во время компиляции (см. приведенную ниже таблицу).

Символ *cpp*	Присваимое значение
STDC	1 — если компилятор соответствует ANSI C; в противном случае — 0
LINE	Номер физической строки исходного файла, в которой находится данный символ
FILE	Имя модуля, который содержит данный символ
DATE	Дата компиляции модуля, содержащего этот символ
TIME	Время компиляции модуля, содержащего данный символ

Использование этих символов иллюстрируется программой test_ansi_c.c, приведенной ниже:

```
#include <stdio.h>
int main()
{
#if __STDC__ == 0 && !defined(__cplusplus)
   printf("cc is not ANSI C compliant\n");
#else
   printf(" %s compiled at %s:%s. This statement is at line %d\n",
      __FILE__, __DATE__, __TIME__, __LINE__);
#endif
   return 0;
}
```

Отметим, что C++ поддерживает символы _LINE_, _FILE_, _DATE_ и _TIME_, но не поддерживает _STDC_.

Наконец, в ANSI C определяется набор стандартных библиотечных функций и соответствующих файлов заголовков. Файлы заголовков являются подмножеством библиотек C, имеющихся в большинстве систем, где реализован K&R C. Стандартные библиотеки ANSI C описаны в главе 4.

1.2. Стандарт ANSI/ISO C++

В начале 80-х годов сотрудник фирмы AT&T Bell Laboratories Бьярне Строуструп (Bjarne Stroustrup) разработал язык программирования C++. Язык C++ был построен на базе C и включал объектно-ориентированные конструкции, такие как классы, производные классы и виртуальные функции, заимствованные из языка simula67 [1]. Целью разработки C++ было "ускорить написание хороших программ и сделать этот процесс более приятным для каждого отдельно взятого программиста" [2]. Название C++, которое придумал Рик Маскитти (Rick Mascitti) в 1983 году, отражает факт происхождения этого языка от C.

С момента своего появления C++ завоевал широкое признание среди профессиональных разработчиков программного обеспечения. В 1989 году Бьярне Строуструп опубликовал руководство *The Annotated C++ Reference Manual* [3], послужившее основой для проекта стандарта ANSI C++, который разработан комитетом ANSI X3J16. В начале 90-х годов к работе этого комитета подключился комитет WG21 Международной организации стандартизации (ISO), и была начата работа по созданию единого стандарта ANSI/ISO C++. Проект такого стандарта опубликован в 1994 году [4], однако стандарт все еще находится в стадии разработки, т.е. официального статуса он не получил.

Большинство новейших коммерческих компиляторов C++, построенных на базе языка AT&T C++ версии 3.0 и выше, соответствуют проекту стандарта ANSI/ISO. В частности, эти компиляторы поддерживают классы C++, производные классы, виртуальные функции, перегрузку операций. Кроме того, они должны поддерживать шаблоны классов, шаблоны функций, обработку исключительных ситуаций и классы потоков ввода-вывода.

Особенности языка C++ описываются в этой книге в соответствии с проектом стандарта ANSI/ISO.

1.3. Различия между ANSI C и C++

Язык C++ требует, чтобы все функции объявлялись или определялись до того, как программа к ним обратится. В ANSI C для тех функций, к которым программа обращается до их объявления и определения, используется стандартное объявление K&R C.

Чтобы показать еще одно различие между ANSI C и C++, рассмотрим объявление функции

```
int foo ();
```

ANSI C трактует его как объявление функции C в старом стиле, интерпретируя ее следующим образом:

```
int foo (...);
```

Это означает, что *foo* можно вызывать с любым числом аргументов. В C++ это объявление трактуется по-другому:

```
int foo (void);
```

Другими словами, *foo* при вызове не может принимать никаких аргументов.

Наконец, в C++ внешние функции компонуются с сохранением типов. Благодаря этому внешняя функция, неверно объявленная и вызванная в программе, заставит редактор связей (*/bin/ld*) сообщить о неопределенном имени функции. В ANSI C метод компоновки с сохранением типов не применяется, поэтому пользовательские ошибки такого рода не вылавливаются.

Между ANSI C и C++ есть много других различий, но с перечисленными выше пользователи сталкиваются чаще всего. (Наиболее подробно язык ANSI C документирован в [5].)

В следующем разделе описываются стандарты POSIX, которые достаточно полно соответствуют потребностям разработчиков UNIX-систем.

1.4. Стандарты POSIX

Поскольку сегодня существует множество версий ОС UNIX и в каждой из них имеется свой набор функций интерфейсов прикладного программирования (API), то системным разработчикам трудно создавать приложения, которые можно было бы легко переносить в другие версии UNIX. Чтобы решить эту проблему, IEEE в 80-х годах сформировал специальную рабочую группу под названием POSIX, перед которой была поставлена задача создания комплекта стандартов, предназначенных для согласования между собой различных реализаций операционных систем. В POSIX имеется несколько подгрупп, в частности, POSIX.1, POSIX.1b и POSIX.1c, которые занимаются разработкой этого комплекта стандартов для системных программистов.

В частности, комитет POSIX.1 предлагает стандарт на базовый интерфейс прикладного программирования операционных систем, в котором определяются API для манипулирования файлами и процессами. Официально он известен как стандарт IEEE 1003.1-1990 [6] и принят также ISO как международный стандарт ISO/IEC 9945:1:1990. Комитет POSIX.1b предлагает набор стандартных API для интерфейса операционных систем реального времени, включающий API межпроцессного взаимодействия (официально известен как стандарт IEEE 1003.4-1993 [7]). Наконец, стандарт POSIX 1.c [8] определяет интерфейс многопоточного программирования. Этот самый новый стандарт POSIX рассматривается в последней главе нашей книги.

Хотя в основе своей работа комитетов POSIX строится на использовании ОС UNIX, предлагаемые ими стандарты предназначены для некой обобщенной операционной системы, которая вовсе не обязательно должна быть системой UNIX. Например, операционные системы VMS (разработчик — Digital Equipment Corporation), OS/2 (International Business Machines) и Windows NT (Microsoft Corporation) совместимы с POSIX, но UNIX-системами не являются. Большинство существующих в настоящее время систем

UNIX, такие как UNIX System V release 4, BSD UNIX 4.4 и операционные системы фирм-производителей UNIX (например, Solaris 2.x от Sun Microsystems, HP-UX 9.05 и 10.x фирмы Hewlett Packard, AIX 4.1.x фирмы IBM и др.), совместимы с POSIX.1, но все равно имеют собственные API.

В этой книге рассматриваются интерфейсы прикладного программирования POSIX.1, POSIX.1b и POSIX.1c, а также API, характерные для систем UNIX. Заметим при этом, что если слово *POSIX* в тексте упоминается без "расширения", оно относится к обоим стандартам — POSIX.1 и POSIX.1b.

Чтобы обеспечить соответствие своей программы стандарту POSIX.1, пользователь должен либо определить в начале каждого исходного модуля программы (до включения заголовков) макрос _POSIX_SOURCE:

```
#define _POSIX_SOURCE
```

либо указать при компиляции для компилятора C++ (CC) опцию -D_POSIX _SOURCE:

```
% CC -D_POSIX_SOURCE *.C
```

С помощью этой константы препроцессор *cpp* отфильтровывает из применяемых пользовательской программой заголовков все коды, не соответствующие стандартам POSIX.1 и ANSI C (например, функции, типы данных и макроопределения). Таким образом, пользовательская программа, которая компилируется с этим ключом и успешно выполняется, соответствует стандарту POSIX.1.

В POSIX.1b для проверки пользовательских программ на соответствие данному стандарту определяется другой макрос. Этот макрос называется _POSIX_C_SOURCE, и его значением является код, обозначающий ту версию POSIX, которой соответствует данная пользовательская программа. Возможные значения макроса _POSIX_C_SOURCE указаны ниже.

Значение _POSIX_C_SOURCE	Соответствие
198808L	Первой версии стандарта POSIX.1
199009L	Второй версии стандарта POSIX.1
199309L	Стандартам POSIX.1 и POSIX.1b

Каждое значение _POSIX_C_SOURCE содержит значение года и месяца, в котором данный стандарт POSIX был утвержден IEEE. Суффикс L в значении показывает, что тип этого значения — длинное целое.

Макрос _POSIX_C_SOURCE можно использовать вместо _POSIX_SOURCE. Однако в некоторых системах, поддерживающих только POSIX.1, макрос _POSIX_C_SOURCE не определен. Поэтому читателям следует просмотреть файл заголовков <unistd.h> у себя в системе, с тем чтобы установить, какой макрос используются в этом файле (возможно, оба).

Есть также макрос _POSIX_VERSION, определяемый, как правило, в заголовке <unistd.h>. Этот макрос содержит версию POSIX, которой

соответствует данная система. В приведенном ниже примере программы проверяется и выводится значение, присвоенное макросу _POSIX_VERSION системы, где эта программа выполняется:

```
/* show_posix_ver.C*/
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include <iostream.h>
#include <unistd.h>

int main()
{
#ifdef _POSIX_VERSION
    cout << "System conforms to POSIX: " << _POSIX_VERSION << endl;
#else
    cout << "_POSIX_VERSION is undefined\n";
#endif
    return 0;
}
```

Вообще говоря, пользовательская программа, которая должна строго соответствовать стандартам POSIX.1 и POSIX.1b, может включать в себя следующие строки:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include <unistd.h>
/* здесь включить другие заголовки */
int main()
{
   ...
}
```

1.4.1. Среда POSIX

Хотя стандарт POSIX разработан на основе UNIX, система, соответствующая этому стандарту, не обязательно является UNIX-системой. В стандартах POSIX несколько соглашений, принятых в UNIX, имеют другой смысл. В частности, большинство стандартных файлов заголовков C и C++ в любой UNIX-системе хранятся в каталоге */usr/include*, и каждый из них указывается в программе следующим образом:

```
#include <имя_файла_заголовков>
```

Такой метод обращения к файлам заголовков принят и в POSIX. Однако в POSIX-совместимой системе не обязательно должен существовать физический файл с именем, заданным оператором *#include*. Данные, которые должны содержаться в этом именованном объекте, могут быть встроены в компилятор или храниться в данной системе каким-нибудь другим способом. В среде POSIX файлы, указываемые с помощью директивы #include, называются не *файлами заголовков*, а просто *заголовками*. В остальной части

книги будет использоваться именно такое правило именования. Кроме того, в POSIX-совместимой системе существование каталога */usr/include* не является обязательным. Если пользователи работают в POSIX-совместимой системе, которая не относится к системам UNIX, то стандартное местонахождение заголовков можно определить по руководству программиста C и C++.

Еще одно различие между POSIX и UNIX состоит в концепции *привилегированного пользователя*. В UNIX привилегированный пользователь имеет право доступа ко всем системным ресурсам и функциям. Его пользовательский идентификатор всегда равен 0. Стандарты POSIX не требуют, чтобы POSIX-совместимые системы соблюдали это правило, и пользовательский идентификатор 0 не означает никаких особых привилегий. Более того, хотя для некоторых API POSIX.1 и POSIX.1b необходимо выполнение функций "с особыми привилегиями", решение о том, как "особые привилегии" будут присваиваться процессу, принимает конкретная система.

1.4.2. Макрокоманды тестирования характеристик по стандарту POSIX

Некоторые возможности ОС UNIX реализуются в POSIX-совместимых системах по усмотрению разработчика. Так, в POSIX.1 определен набор макрокоманд тестирования характеристик, который (если он определен) позволяет установить, реализованы ли в конкретной системе соответствующие возможности.

Эти макрокоманды, если они определены, находятся в заголовке <unistd.h>. Ниже перечислены их имена и выполняемые действия.

Макрокоманда тестирования характеристик	Действие
_POSIX_JOB_CONTROL	Система поддерживает BSD-подобное управление заданиями
_POSIX_SAVED_IDS	Каждый процесс, работающий в системе, хранит установленные идентификаторы пользователя и группы и имеет возможность изменять их значения посредством API *seteuid* и *setegid*
_POSIX_CHOWN_RESTRICTED	Если определено значение -1, пользователи имеют право изменять принадлежность файлов, которыми владеют. В противном случае только пользователи с особыми привилегиями могут изменять принадлежность всех файлов в системе. Если этот макрос в заголовке <unistd.h> не определен, пользователи должны с помощью функции *pathconf* или функции *fpathconf* (см. следующий раздел) проверить наличие разрешения на изменение принадлежности для каждого файла

Макрокоманда тестирования характеристик	Действие
_POSIX_NO_TRUNC	Если определено значение -1, то все длинные путевые имена, переданные в API, усекаются до NAME_MAX байтов; в противном случае выдается сообщение об ошибке. Если этот макрос в заголовке <unistd.h> не определен, пользователи должны с помощью функции *pathconf* или функции *fpathconf* (см. следующий раздел) проверить режим усечения путевого имени для каждого каталога
_POSIX_VDISABLE	Значение этого макроса используется как запрещающий символ для специальных управляющих символов терминальных устройств ввода-вывода. Если определено значение -1, то запрещающего символа нет. Если этот макрос в заголовке <unistd.h> не определен, пользователи должны с помощью функции *pathconf* или функции *fpathconf* (см. следующий раздел) проверить наличие запрещающего символа для каждого файла терминального устройства

В приведенном ниже примере на экран выводятся определенные стандартом POSIX параметры конфигурации, поддерживаемые в любой UNIX-системе:

```
/* show_test_macros.C */
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE     199309L
#include <iostream.h>
#include <unistd.h>

int main()
{
#ifdef _POSIX_JOB_CONTROL
    cout << "System supports job control\n";
#else
    cout << "System does not support job control\n";
#endif

#ifdef _POSIX_SAVED_IDS
    cout << "System supports saved set-UID and saved set-GID\n";
#else
    cout << "System does not support saved set-UID and saved"
            <<"set-GID\n";
#endif

#ifdef _POSIX_CHOWN_RESTRICTED
    cout << "chown restricted option is: " <<
                _POSIX_CHOWN_RESTRICTED << endl;
#else
    cout << "System does not support system-wide chown_restricted option\n";
#endif
```



```
#ifdef _POSIX_NO_TRUNC
    cout << "Pathname trucnation option is: " <<  _POSIX_NO_TRUNC << endl;
#else
    cout << "System does not support system-wide pathname"
         <<"trucnation option\n";
#endif

#ifdef _POSIX_VDISABLE
    cout << "Disable character for terminal files is: "
         <<_POSIX_VDISABLE << endl;
#else
    cout << "System does not support _POSIX_VDISABLE\n";
#endif
    return 0;
}
```

1.4.3. Проверка ограничений во время компиляции и во время выполнения

В стандартах POSIX.1 и POSIX.1b определен набор ограничений, задающих предельные значения параметров конфигурации системы. Эти ограничения устанавливаются с помощью макросов в заголовке <limits.h>. Многие из них взяты из систем UNIX и имеют те же имена, что и их UNIX-эквиваленты, но с префиксом _POSIX_. Например, в системах UNIX определена константа CHILD_MAX, ограничивающая максимальное число порожденных процессов, которые процесс может создать в любой заданный момент времени. В POSIX.1 этой константе соответствует константа _POSIX_CHILD_MAX. Причина определения этих констант состоит в том, что, несмотря на наличие в большинстве UNIX-систем похожего набора констант, их значения в разных UNIX-системах существенно различаются. Константы, определенные в POSIX, задают минимальные значения параметров для всех соответствующих стандарту POSIX систем; в силу этого прикладные программисты при разработке программ закладывают в них свои ограничения на параметры конфигурации систем.

Ниже приведен перечень констант, определенных стандартом POSIX.1 в заголовке <limits.h>.

Ограничение	Минимальное значение	Смысл
_POSIX_CHILD_MAX	6	Максимальное число порожденных процессов, которые процесс может создать в любой момент времени
_POSIX_OPEN_MAX	16	Максимальное число файлов, которые процесс может открыть одновременно
_POSIX_STREAM_MAX	8	Максимальное число потоков ввода-вывода, которые процесс может открыть одновременно

Ограничение	Минимальное значение	Смысл
_POSIX_ARG_MAX	4096	Максимальный размер (в байтах) аргументов, которые могут быть переданы при вызове функции *exec*
_POSIX_NGROUP_MAX	0	Максимальное число дополнительных групп, которым может принадлежать процесс
_POSIX_PATH_MAX	255	Максимально допустимое число символов в путевом имени
_POSIX_NAME_MAX	14	Максимально допустимое число символов в имени файла
_POSIX_LINK_MAX	8	Максимальное число ссылок на файл
_POSIX_PIPE_BUF	512	Максимальный размер блока данных, который может быть прочитан из канала или записан в канал в ходе одного обращения к нему
_POSIX_MAX_INPUT	255	Максимальная емкость (в байтах) входного буфера терминала
_POSIX_MAX_CANON	255	Максимальное количество байтов в строке в каноническом режиме ввода*
_POSIX_SSIZE_MAX	32767	Максимальное значение, которое может храниться в объекте типа *ssize_t*
_POSIX_TZNAME_MAX	3	Максимальное количество символов в названии часового пояса

* Канонический режим ввода предусматривает наличие функций редактирования строки *erase* (удалить символ) и *kill* (удалить строку), а также ограничение вводимой строки символами EOL, EOF, NL. — *Прим. ред.*

Ниже перечислены константы, определенные в стандарте POSIX.1b.

Ограничение	Минимальное значение	Смысл
_POSIX_AIO_MAX	1	Максимально допустимое количество одновременных операций асинхронного ввода-вывода
_POSIX_AIO_LISTIO_MAX	2	Максимальный размер списка операций ввода-вывода, возвращаемого системным вызовом *listio*
_POSIX_TIMER_MAX	32	Максимальное количество таймеров, которые могут одновременно использоваться процессом.
_POSIX_DELAYTIMER_MAX	32	Максимально допустимое количество переполнений для каждого таймера

Ограничение	Минимальное значение	Смысл
_POSIX_MQ_OPEN_MAX	2	Максимальное количество очередей сообщений, которые могут быть одновременно доступны одному процессу
_POSIX_MQ_PRIO_MAX	2	Максимальное значение приоритета, который может быть назначен сообщению
_POSIX_RTSIG_MAX	8	Максимальное количество сигналов реального времени, зарезервированных для прикладных программ
_POSIX_SIGQUEUE_MAX	32	Максимальное количество сигналов реального времени, которые процесс может поставить в очередь в любой момент времени
_POSIX_SEM_NSEMS_MAX	256	Максимальное количество семафоров, которые могут одновременно использоваться процессом
_POSIX_SEM_VALUE_MAX	32767	Максимальное значение, которое может быть присвоено семафору

Следует отметить, что константы, определенные в стандартах POSIX, задают лишь минимальные значения параметров конфигурации системы. Любую POSIX-совместимую систему можно конфигурировать с более высокими значениями этих пределов. Более того, не все эти константы нужно указывать в заголовке <limits.h>, потому что некоторые из лимитов могут быть неопределенными или изменяться в зависимости от конкретного файла.

Чтобы определить фактически установленные ограничения конфигурации для всей системы или для отдельных ее объектов, можно запросить значения этих ограничений с помощью определенных стандартом POSIX.1 функций *sysconf, pathconf* и *fpathconf*. Функция *sysconf* служит для запроса общесистемных ограничений конфигурации, которые установлены в данной системе; функции *pathconf* и *fpathconf* используются для запроса ограничений конфигурации, связанных с файлами. Единственное различие этих функций состоит в том, что *pathconf* в качестве аргумента принимает путевое имя файла, тогда как *fpathconf* — дескриптор файла. Вот прототипы этих функций:

```
#include <unistd.h>

long sysconf ( const int limit_name );
long pathconf ( const char* pathname, int flimit_name );
long fpathconf ( const int fdesc, int flimit_name );
```

Значение аргумента *limit_name* — макрос, определенный в заголовке <unistd.h>. Ниже перечислены возможные значения *limit_name* и данные, возвращаемые функцией *sysconf*.

Значение ограничения	Возвращаемые данные
_SC_ARG_MAX	Максимальный размер (в байтах) значений аргументов, которые могут быть переданы в вызов API *exec*
_SC_CHILD_MAX	Максимальное число порожденных процессов, которыми процесс может владеть одновременно
_SC_OPEN_MAX	Максимальное число открытых файлов на один процесс
_SC_NGROUPS_MAX	Максимальное число дополнительных групп на один процесс
_SC_CLK_TCK	Количество тактов системных часов в секунду
_SC_JOB_CONTROL	Значение _POSIX_JOB_CONTROL
_SC_SAVED_IDS	Значение _POSIX_SAVED_IDS
_SC_VERSION	Значение _POSIX_VERSION
_SC_TIMERS	Значение _POSIX_TIMERS
_SC_DELAYTIMER_MAX	Максимальное количество переполнений на один таймер
_SC_RTSIG_MAX	Максимальное количество сигналов реального времени
_SC_MQ_OPEN_MAX	Максимальное количество очередей сообщений на один процесс
_SC_MQ_PRIO_MAX	Максимальное значение приоритета, который может быть назначен сообщению
_SC_SEM_NSEMS_MAX	Максимальное количество семафоров на один процесс
_SC_SEM_VALUE_MAX	Максимальное значение, которое может быть присвоено семафору
_SC_SIGQUEUE_MAX	Максимальное количество сигналов реального времени, которые процесс может поставить в очередь в любой момент времени
_SC_AIO_LISTIO_MAX	Максимальный размер списка операций ввода-вывода, возвращаемого системным вызовом *listio*
_SC_AIO_MAX	Максимально допустимое количество одновременных операций асинхронного ввода-вывода

Как видите, все макросы, используемые в качестве значения аргумента *sysconf*, имеют префикс _SC_. Аналогичным образом значение аргумента

flimit_name — это макрос, определенный в заголовке <unistd.h>. Все эти константы имеют префикс _PC_. Ниже перечислены некоторые из них, а также значения, которые функции *pathconf* и *fpathconf* возвращают для указанного объекта.

Значение ограничения	Возвращаемые данные
_PC_CHOWN_RESTRICTED	Значение _POSIX_CHOWN_RESTRICTED
_PC_NO_TRUNC	Возвращает значение _POSIX_NO_TRUNC
_PC_VDISABLE	Возвращает значение _POSIX_VDISABLE
_PC_PATH_MAX	Максимальная длина путевого имени в байтах
_PC_LINK_MAX	Максимальное число ссылок на файл
_PC_NAME_MAX	Максимально допустимая длина имени файла в байтах
_PC_PIPE_BUF	Максимальный размер блока данных, который может быть автоматически прочитан из канала или записан в канал
_PC_MAX_CANON	Максимальный размер (в байтах) строки в каноническом режиме ввода
_PC_MAX_INPUT	Максимальная емкость (в байтах) входного буфера терминала

Эти переменные дублируют соответствующие переменные, определенные в большинстве UNIX-систем (имена UNIX-переменных совпадают с POSIX-именами, за исключением префикса _POSIX_). Указанные переменные можно использовать в программах, поскольку во время компиляции для них будут выполнены макроподстановки:

```
char pathname [_POSIX_PATH_MAX + 1 ];
for (int i=0; i<_POSIX_OPEN_MAX; i++)
   close(i);              // закрыть все дескрипторы файлов
```

Использование функций *sysconf*, *pathconf* и *fpathconf* иллюстрируется на примере следующей программы *test_config.C*:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include <stdio.h>
#include <iostream.h>
#include <unistd.h>

int main()
{
    int res;
    if ((res=sysconf(_SC_OPEN_MAX))==-1)
        perror("sysconf");
    else
        cout << "OPEN_MAX: " << res << endl;
```

```
    if ((res=pathconf("/",_PC_PATH_MAX))==-1)
        perror("pathconf");
    else
       cout << "Max path name: " << (res+1) << endl;

    if ((res=fpathconf(0,_PC_CHOWN_RESTRICTED))==-1)
        perror("fpathconf");
    else
       cout << "chown_restricted for stdin: " << res << endl;

    return 0;
}
```

1.5. Стандарт POSIX.1 FIPS

FIPS — это Federal Information Processing Standard (федеральный стандарт на обработку информации). Стандарт POSIX.1 FIPS был разработан Национальным институтом стандартов и технологий США (NIST, ранее — Национальное бюро стандартов), который подчиняется Министерству торговли США. Последний вариант этого стандарта, FIPS 151-1, построен на основе стандарта POSIX.1-1988. Стандарт POSIX.1 FIPS — это руководящий документ для федеральных ведомств, приобретающих компьютерные системы. В частности, согласно стандарту FIPS, являющемуся ограниченным вариантом стандарта POSIX.1-1988, все FIPS-совместимые системы должны соответствовать следующим требованиям:

- Управление заданиями; должен быть определен макрос _POSIX_JOB _CONTROL.
- Поддержка сохранения установленных UID и GID; должен быть определен макрос _POSIX_SAVED_IDS.
- Отсутствие поддержки длинных путевых имен; должен быть определен макрос _POSIX_NO_TRUNC с любым значением, кроме -1.
- Должен быть определен макрос _POSIX_CHOWN_RESTRICTED с любым значением, кроме -1. Это значит, что изменять принадлежность файлов во всей системе может только уполномоченный на это пользователь.
- Должен быть определен макрос _POSIX_VDISABLE с любым значением, кроме -1.
- Значение символа NGROUP_MAX должно быть равно минимум 8.
- API чтения и записи должны возвращать число байтов, передаваемых после того, как эти API прерываются сигналами.
- Идентификатор группы вновь созданного файла должен наследовать идентификатор группы каталога, который его содержит.

Стандарт FIPS содержит больше ограничений, чем стандарт POSIX.1, поэтому система, соответствующая FIPS 151-1, соответствует и POSIX.1-1988 (но не наоборот). По сравнению с последней версией POSIX.1 стандарт FIPS устарел и используется главным образом федеральными ведомствами США. По этой причине мы акцентируем внимание не на FIPS, а на POSIX.1.

1.6. Стандарты X/Open

Организация X/Open была создана группой европейских компаний с целью разработки общего интерфейса операционных систем для производимых ими компьютеров. В 1989 году организация издала выпуск 3 руководства по разработке переносимого программного обеспечения *X/Open Portability Guide* (XPG3), а в 1994 году — выпуск 4. В этих документах определяется набор общих средств и функций интерфейсов прикладного программирования языка C, которые должны присутствовать во всех "открытых системах" на базе UNIX. Руководства XPG3 [9] и XPG4 [10] основаны на стандартах ANSI C, POSIX.1, POSIX.2 и содержат дополнительные конструкции, разработанные организацией X/Open.

Кроме того, в 1993 году несколько фирм-производителей компьютеров (Hewlett-Packard, International Business Machines, Novell, Open Software Foundation, Sun Microsystems) инициировали проект под названием Common Open Software Environment (COSE, общая открытая программная среда). Целью проекта было разработать единую спецификацию интерфейса программирования для UNIX, которую поддерживали бы все фирмы. Эта спецификация известна как Spec 1170 и включена в XPG4 как часть спецификаций Common Application Environment (CAE, общая прикладная среда) организации X/Open.

Спецификации CAE имеют гораздо более широкую область применения, чем стандарты POSIX и ANSI C. Это значит, что приложения, которые соответствуют X/Open, обязательно соответствуют и стандартам ANSI C и POSIX (но не наоборот). Кроме того, хотя большинство производителей компьютеров и независимых продавцов программного обеспечения приняли POSIX и ANSI C, некоторым из них еще предстоит обеспечить соответствие стандартам X/Open. Учитывая все это, мы акцентируем внимание в основном на общем для всех систем интерфейсе системного программирования для UNIX и стандартах ANSI C, POSIX. Подробную информацию о спецификациях X/Open CAE читатели смогут получить из [4,5].

1.7. Заключение

В этой главе дан обзор различных стандартов, которые имеют важное значение для системных программистов, разрабатывающих программное обеспечение для UNIX. Цель обзора — ознакомить читателей с этими стандартами, помочь понять их важность. Детали упомянутых стандартов, соответствующие им функции и API, имеющиеся в большинстве UNIX-систем, рассматриваются в последующих главах книги.

1.8. Литература

1. O-J. Dahl, B. Myrhaug, and K. Nygaard, *SIMULA Common Base Language*, 1970.
2. Bjarne Stroustrup, *The C++ Programming Language*, Second Edition, 1991.
3. Margaret A. Ellis, and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.
4. Andrew Koenig, *Working Paper for Draft Proposed International Standard for* Information Systems — Programming Language C++ (Committees: WG21/NO414, *X3J16/94-0025*), 1994.
5. American National Standard Institute, *American National Standard for Information Systems — Programming Language C, X3.159 — 1989*, 1989.
6. Institute of Electrical and Electronics Engineers, *Information Technology — Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C language], IEEE 1003.1*, 1990.
7. Institute of Electrical and Electronics Engineers, *Information Technology — Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C language] — Amendment: Real-Time Extension, IEEE 1003.1b*, 1993.
8. Institute of Electrical and Electronics Engineers, *Information Technology — Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C language] — Amendment: Thread Extension, IEEE 1003.1c*, 1995.
9. X/Open, *X/Open Portability Guide*, Prentice Hall, 1989.
10. X/Open, *X/Open CAE Specification, Issue 4*, Prentice Hall, 1994.

ГЛАВА 2

Обзор языка C++

Эта глава содержит обзор основных конструкций языка C++, который соответствует проекту стандарта ANSI/ISO [1]. Предполагается, что читатели знакомы с языком C++, хотя бы на начальном уровне. В главе дается беглый обзор методов программирования на C++, что позволит освежить их в памяти читателя. Описываются новые особенности языка, определенные стандартом ANSI/ISO,— шаблоны классов, а также методы обработки исключительных ситуаций. Читатели, которым нужны более подробные справки по этому языку программирования, могут обратиться к [2, 3].

Помимо описания конструкций языка C++ в этой главе рассматриваются также стандартные классы ввода-вывода и методы объектно-ориентированного программирования. Стандартные классы ввода-вывода — очень мощное и функционально богатое средство. Они фактически заменяют функции потокового ввода-вывода и строковые функции языка C. Знание классов ввода-вывода позволит добиться максимально высокой степени многократного использования кода, сократить время и стоимость разработки приложений.

Методы объектно-ориентированного программирования позволяют перейти от алгоритмических моделей программ к объектным. При объектно-ориентированном программировании пользователей в первую очередь заботят типы объектов, с которыми приходится иметь дело их программам, свойства этих объектов, а также то, как они взаимодействуют между собой и с другими пользователями. Методы объектно-ориентированного программирования ценны для приложений баз данных и графических пользовательских интерфейсов. Кроме того, они позволяют скрыть детали низкоуровневого сетевого коммуникационного протокола за счет формирования высокоуровневого интерфейса для разработчиков сетевых приложений. В последующих главах книги будет показано, как это делается.

2.1. Средства объектно-ориентированного программирования в C++

C++ поддерживает объявления классов. Классы используются для построения определяемых пользователем типов данных. В каждом классе распределяется память для хранения данных и устанавливаются допустимые операции для каждого объекта данных этого типа. Поэтому программисты, работающие на C++, тратят меньше времени на традиционное алгоритмическое проектирование своих приложений, концентрируя усилия на создании классов, управлении объектами классов и их взаимодействием.

Класс позволяет делать недоступными внутренние данные, представляя их как открытые (public), закрытые (private) и защищенные (protected). Открытые данные класса доступны для любых пользовательских функций, не определенных внутри класса. Закрытые данные доступны только для функций-членов, определенных в этом классе. Наконец, защищенные данные класса являются закрытыми для всех пользовательских функций, но открытыми для всех функций-членов класса и функций-членов подклассов. Такая сложная схема доступа имеет целью позволить разработчикам управлять доступом к данным объектов различных классов и манипулировать ими. Однако она не дает возможности изменять данные объектов класса где-нибудь в пользовательской программе. Более того, любые изменения закрытых и защищенных данных класса будут в минимальной степени влиять на пользовательские функции при условии, что функции-члены, применяемые при обращении к этим данным, останутся неизменными.

Класс устанавливает четко определенный интерфейс для взаимодействия объектов этого типа с остальным миром. Это дает возможность пользователям изменять внутреннюю реализацию любого класса, сохраняя при этом всю остальную программу в рабочем состоянии (если интерфейс класса не изменяется). В результате можно получить оптимально построенные программы на C++, легкие для сопровождения и модернизации.

Еще одно преимущество классов состоит в том, что они содействуют совместному использованию кода. В частности, можно породить новый класс от одного или нескольких существующих классов, и этот новый класс будет содержать всю память для хранения данных и все функции того класса (или классов), из которого он был создан. Более того, в этом новом классе могут определяться дополнительные данные и функции, уникальные для объектов этого нового типа, и даже переопределяться функции, которые этот класс наследует от своего базового класса (или классов). Таким образом, наследование классов дает максимальную гибкость в формировании новых классов, похожих на существующие, но не идентичных им.

Как и другие объектно-ориентированные языки, C++ поддерживает для классов *функцию-конструктор* и *функцию-деструктор*. Эти функции обеспечивают надлежащую инициализацию объектов при их создании и удаление данных при их уничтожении. Кроме того, в C++ для выделения и освобождения динамической памяти объектам определены операции *new* и *delete*. Однако

в отличие от других объектно-ориентированных языков здесь нет встроенных средств сбора мусора (garbage), предназначенных для управления используемой объектами динамической памятью, а конструктор и деструктор не обязательны для всех определенных классов. Эти послабления сделаны для уменьшения издержек, отрицательно влияющих на производительность программ C++, но при этом требуется, чтобы разработчики были очень внимательны при создании своих программ.

Стандарт ANSI/ISO C++ поддерживает шаблоны классов и функций. Шаблоны позволяют пользователям создавать и отлаживать некоторые родовые (generic) классы и функции. На основе шаблонов могут быть заданы реальные классы и функции, работающие с различными типами данных. Это значительно экономит время разработки и отладки программ. Помимо этого, в C++ определен формальный метод обработки исключительных ситуаций, которые могут возникнуть в программах. Для всех приложений C++ предусмотрены единые методы такой обработки.

Подведем итоги и сформулируем основные цели объектно-ориентированного программирования:

- абстрактное представление данных с целью создания четко определенного интерфейса для всех объектов;
- наследование классов, способствующее многократному использованию кода;
- полиморфизм, при котором классы, производные от других классов, могут содержать данные и функции, отличные от тех, которые определены в порождающих классах;
- моделирование объектов и их взаимодействия в реальных ситуациях.

C++ позволяет достичь всех этих целей. Кроме того, он совместим снизу вверх с языком C. Поэтому программисты, пишущие на C, могут начать использовать C++ совместно со своими C-программами, а затем в удобном для себя темпе перейти полностью на C++ и объектно-ориентированные конструкции.

2.2. Объявление классов в C++

Класс в C++ служит для абстрактного представления данных разных типов. Он состоит из данных-членов и функций-членов. Они, в свою очередь, могут классифицироваться как закрытые, открытые и защищенные. Закрытые данные-члены и функции-члены доступны только через функции-члены этого же класса, тогда как открытые доступны для любых объектов. Открытые члены образуют для объектов данного класса интерфейсы, с помощью которых к ним можно обращаться из "внешнего мира". Защищенные данные-члены и функции-члены похожи на закрытые, но доступны также для функций-членов подклассов.

Если ссылка на данное-член или функцию-член класса дается за пределами места объявления класса, необходимо уточнить их имена с помощью

операции расширения области видимости "::". Имя, стоящее слева от знака операции "::",— это имя класса, а имя, стоящее справа от него, — переменная или функция, определенная в классе. Например, *menu::num_fields* обозначает данное-член *num_fields* класса *menu*. Если слева от знака операции расширения области видимости имени нет, то имя, указанное справа, представляет собой глобальную переменную или функцию. Например, если есть глобальная переменная *x*, а в объявлении класса есть данное-член *x*, то функции-члены этого класса могут обращаться к глобальной переменной *x* посредством записи *::x*, а к данному-члену *x* — с использованием записи *x* или записи *<имя_класса>::x*.

В следующем заголовке *menu.h* объявляется класс *menu*:

```
#ifndef MENU_H
#define MENU_H

class menu
{
   private:
      char* title;
   protected:
      static int num_fields;
   public:
     // функция-конструктор
     menu( const char* str )
     {
        title = new char[strlen(str)+1];
        strcpy(title,str);
        num_fields = 0;
     };
     // функция-конструктор
     menu()
     {
        title = 0;
        num_fields = 0;
     };
     // функция-деструктор
     ~menu()
     {
        delete title;
     };
     void incr_field( int size=1 )
     {
        num_fields+=size;
     };
     static int fields()
     {
        return num_fields;
     };
     char* name()
     {
        return title;
     };
```



```
};
#endif          /*menu.h*/
```

В данном примере члены *menu::num_fields* и *menu::fields()* объявлены как статические. При таком объявлении объекты получают доступ к одному экземпляру каждого статического данного-члена. В противном случае каждый объект получает собственную копию этих членов. Статические данные-члены используются как глобальные переменные всеми объектами этого же типа. Доступность статических данных-членов для объектов других классов определяется тем, как они объявлены: как закрытые, защищенные или открытые.

Статические данные-члены, если они используются в программе, должны определяться в исходном модуле. Рассмотрим в качестве примера определение данного-члена *menu::num_fields* с нулевым начальным значением:

```
//имя модуля: a.C
#include <string.h>
#include "menu.h"
int menu::num_fields = 0;
int main()   {...}
```

Хотя здесь *menu::num_fields* — защищенное данное-член, его можно определить и инициализировать в области видимости программы. Однако дальнейшее изменение этой переменной в пользовательских программах должно производиться только через функции-члены класса *menu*, функции подклассов или дружественные функции. Это же правило касается закрытых статических данных-членов.

Статические функции-члены могут обращаться только к статическим данным-членам в классе. Так, в классе *menu* функция *menu::fields()* не может обращаться к данному-члену *menu::title*. В то время как нестатические функции-члены должны вызываться через объекты класса, статические функции такого ограничения не имеют:

```
menu abc ("Example");
abc.incr_field( 5 );
cout<<"static func. called independent of objects:"
      <<menu::fields()<<endl;
cout<<"Static func. can also be called via object:"
      <<abc.fields()<<endl;
```

Статические данные-члены и функции обычно используются для отслеживания количества созданных объектов того или иного класса, а также для сбора других общих статистических данных. С их помощью можно также управлять динамической памятью, создаваемой всеми объектами класса, и производить уборку мусора.

Функции *menu::menu* — это конструкторы. Функция-конструктор вызывается при создании объекта класса и инициализирует данные-члены вновь созданного объекта. Функцию-конструктор можно перегружать, т.е. в одном классе может быть определено несколько конструкторов с различными

сигнатурами. Например, в представленных ниже определениях объектов используются разные функции-конструкторы:

```
menu abc;              // используется menu::menu()
menu xyz("Example"); // используется menu::menu( const char* str)
```

Функция *menu()::~menu()* является деструктором. Она вызывается тогда, когда объект класса выходит из области видимости, и предназначена для обеспечения завершения обработки данных надлежащим образом (например, освобождения используемой объектом динамической памяти). Функция-деструктор не перегружается и не принимает аргументов.

В примере с классом *menu* все определения функций-членов размещены в объявлении класса. Это значит, что эти функции-члены будут использоваться как встроенные (inline) функции (в языке С аналогом встроенных функций являются макроопределения). Преимущество использования inline-функций состоит в повышении производительности программ благодаря отсутствию дополнительных затрат на вызовы функций. У встроенных функций есть и недостаток: любые изменения в такой функции требуют перекомпиляции исходных модулей, содержащих ссылки на нее.

Пользователь может объявить функции-члены класса как невстроенные, поместив их определения в отдельный исходный модуль. В частности, заголовок *menu.h* можно изменить следующим образом:

```
#ifndef MENU_H
#define MENU_H

class menu
{
   private:
      char* title;
   protected:
      static int num_fields;
   public:
      // конструктор
      menu( const char* str );
         menu();
      // деструктор
      ~menu();
      void incr_field( int size=1 );
      static int fields();
      char* name() ;
};
#endif              /*menu.h*/
```

Тогда для определения функций-членов нужно создать отдельный модуль C++, например, *menu.C*:

```
// source file name: menu.C
#include <string.h>
#include "menu.h"
int  menu::num_fields = 0;
```



```
// конструктор с аргументом
menu::menu ( const char* str )
{
    title = new char[strlen(str)+1];
    strcpy (title, str );
    num_fields = 0;
}
// конструктор без аргумента
menu::menu()
{
    title = 0;
    num_fields = 0;
}
// деструктор
menu::~menu()
{
    delete title;
}
// нестатическая функция-член
void menu::incr_field ( int siz )
{
    num_fields += siz;
}
// статическая функция-член
int menu::fields()
{
    return num_fields;
 }
// еще одна нестатическая функция-член
char* menu::name()
{
    return title;
}
```

Любая программа, которая использует класс *menu* для создания исполняемого объекта, должна компилироваться совместно с файлом *menu.C*. Возьмем, например, файл *test_menu.C*:

```
// source test_menu.C
#include <iostream.h>
#include "menu.h"
int main()
{
   menu abc ("Test");
   cout << abc.name();
   return menu::fields();
}
```

Путем компилирования файла *test_menu.C* создается выполняемая программа *a.out*:

```
% CC test_menu.C menu.C
% a.out
Test
```

В дополнение к сказанному отметим, что в объявлении функции *menu::incr_field* в файле *menu.h* аргумент *size* имеет значение по умолчанию, тогда как в файле *menu.C* в определении функции *menu::incr_field* указывать значение по умолчанию для этого аргумента не разрешается. В C++ 1.0 подобное объявление было возможно, но поскольку его использование может привести к противоречиям в присвоении значений по умолчанию аргументам функций, в последующих версиях оно не допускается.

2.3. Дружественные функции и классы

Используемая в C++ концепция "дружественности" позволяет предоставлять прямой доступ к закрытым и защищенным данным-членам класса определенным функциям и функциям-членам других классов. Этот принцип рассчитан на особые случаи, когда для функции более эффективным является прямой доступ к закрытым данным объекта, нежели доступ через функцию-член его класса. Например, оператор "<<" обычно объявляется как дружественная функция для классов, чтобы можно было выводить на экран объекты этих классов так, как будто они имеют базовые типы данных (например, *int*, *double*).

Поскольку дружественные функции могут непосредственно обращаться к закрытым данным объектов класса и изменять их, компилятору необходимо сообщить о том, что это особые, уполномоченные функции. Поэтому их имена должны быть указаны в объявлении класса с ключевым словом *friend*. Это, кроме того, должно служить напоминанием пользователю о том, что всякий раз при изменении класса может понадобиться соответствующая модификация всех дружественных ему функций.

В приведенном ниже примере демонстрируется использование дружественной функции и дружественного класса:

```
// source module: friend.C
#include <iostream.h>
int year;
class foo;
class dates
{
      friend ostream& operator<<(ostream&,dates&);
      int year, month, day;
   public:
      friend class foo;
      dates() { year=month=day = 0; };
      ~dates() {};
      int sameDay(int d) const { return d==day; };
      void set(int y) const { ::year = y; };
      void set(int y) { year = y; };
};
class foo
{
   public:
      void set(dates& D, int  year) {D.year = year; };
};
```



```
ostream& operator<< (ostream& os, dates& D)
{
    os << D.year << " ," << D.month << " ," << D.day;
    return os;
}

int main()
{
    dates Dobj;
    foo Fobj;
    Fobj.set(Dobj, 1998);
    clog << "Dobj: " << Dobj << '\n';
    return 0;
}
```

В этом примере операция "<<" и класс *foo* объявлены как дружественные классу *dates*. Это значит, что операция "<<" (фактически, это функция-член) и функции-члены класса *foo* могут обращаться к закрытым данным-членам всех объектов класса *dates*. Компиляция и пробный запуск этой программы дают такие результаты:

```
% CC friend.C
% a.out
Dobj: 1998, 0, 0
```

2.4. Функции-члены, объявленные с декларацией const

Функции-члены, объявленные с декларацией *const*, — это особые функции-члены, которые не могут модифицировать данные-члены в своем классе. Они предназначены для размещения объектов класса, которые определены как *const*. Объект, объявленный с декларацией *const*, может вызывать только функции-члены своего класса, также объявленные с декларацией *const*. Это гарантирует невозможность изменения данных объекта.

Если const-объект вызывает функцию-член, объявленную без декларации *const*, компилятор C++ сигнализирует об ошибке. Единственное исключение: к const-объектам можно применять функции-конструкторы и функции-деструкторы. Функции-члены, объявленные как с декларацией *const*, так и без нее, но имеющие однотипную сигнатуру, можно перегружать.

В следующем примере показано, как определяются и используются функции-члены с декларацией *const*:

```
// source module: const.C
#include <iostream.h>
static  int year = 0;
class dates
{
      int year, month, day;
   public:
      dates() { year=month=day = 0; };
```

```
        ~dates() {};
        void set(int y) { year = y; };
        // функции-члены типа const
        void print( ostream& = cerr ) const;
        int sameDay(int d) const { return d==day; };
        // замечание: эта функция перегружена:
        void set(int y) const { ::year = y; };
};

void dates::print( ostream& os ) const
{
    os << year << "," << month << "," << day << '\n';
}

int main()
{
    const dates foo;      // const object
    dates foo1;           // non-const object
    foo.set(1915);        // ::year = 1915
    foo.print();          // year=0, month=0, day=0
    foo1.set(25);         // foo1.year=25
    foo1.print();         // year=25,month=0,day=0
    return 0;
}
```

В этом примере *foo* — const-объект, а *foo1* — нет. Оба объекта инициализируются конструктором *dates::dates*. Оператор *foo.set(1915)* вызывает функцию-член *dates::set* типа *const*, а оператор *foo1.set(25)* — просто функцию-член *dates::set*. Оба оператора могут вызывать функцию-член *dates::print*. Это нормально, так как объекты, не относящиеся к типу *const*, всегда могут вызвать функции-члены типа *const* (но не наоборот).

Компиляция и пробный запуск этой программы дают такие результаты:

```
% CC const.C
% a.out
0, 0, 0
25, 0, 0
```

2.5. Наследование классов в C++

Наследование позволяет порождать класс от одного или нескольких существующих классов. Новый класс называется *подклассом* (или классом-потомком), а класс (или классы), от которого он произведен,— *базовым классом*, *надклассом*, или родительским классом.

Подкласс наследует и может обращаться ко всем защищенным и открытым данным-членам и функциям своего базового класса (классов). Кроме того, в подклассе могут определяться другие данные-члены и функции, уникальные для него самого.

В следующем заголовке *window.h* объявляется класс *window*, который является подклассом класса *menu*:

```
// source file name: window.h
#ifndef WINDOW_H
#define WINDOW_H
#include <iostream.h>
#include "menu.h"

class window : public menu
{
   private:
      int xcord, ycord;
    public:
      // конструктор
       window( const int x, const int y, const char* str ) : menu(str)
       {
         xcord = x;
         ycord = y;
       };
      // деструктор
      ~window() {};
      // функция, предназначенная для window
      void show (ostream& os )
      {
         os << xcord << ',' << ycord << " => " << name() << endl;
      };
};
#endif
```

В объявлении подкласса имя базового класса может предваряться ключевым словом public или private. Это означает, что открытые данные-члены и функции базового класса следует считать в подклассе соответственно открытыми или закрытыми. Если такого ключевого слова нет, по умолчанию они являются закрытыми.

Функция-член подкласса может непосредственно обращаться только к защищенным и открытым данным-членам своего базового класса (классов). Подкласс может явно отмечать определенные открытые или защищенные данные-члены и (или) функции закрытых базовых классов как соответственно открытые или защищенные в этом подклассе.

Например, класс *window2* делает все данные-члены и функции класса *menu*, которые он наследует, закрытыми, за исключением переменной *menu::num_fields*, которая в этом подклассе трактуется как защищенная:

```
class window2:private menu
{
   private:
        int xcord, ycord;
   protected:
        menu::num_fields;      // сделать menu::num_field защищенной
   public:
        ...
);
```

Когда определяется объект подкласса, то функции-конструкторы этого подкласса и его базового класса (классов) вызываются в следующем порядке:

- конструкторы базового класса в последовательности, указанной в объявлении подкласса;
- конструкторы данных-членов в последовательности, объявленной в подклассе;
- конструкторы подкласса.

Так в объявлении подкласса

```
class a,b;
class base1, base2;
class sub:public base1, private base2;
{
     a var1;
     b var2;
  public:
     ....
};
sub foo;
```

порядок вызова функций-конструкторов для переменной *foo* будет следующим: *base1::base1, base2::base2, a::a, b::b* и, наконец, *sub::sub*.

Если объект подкласса находится вне области видимости, то функции-деструкторы этого подкласса и его базового класса (классов) вызываются в таком порядке:

- деструктор подкласса;
- деструкторы данных-членов в последовательности, обратной объявленной в подклассе;
- деструкторы базового класса в последовательности, обратной указанной в объявлении подкласса.

Если в предыдущем примере переменная *foo* выходит из области видимости, то порядок вызова функций-деструкторов будет следующим: *sub::~sub, b::~b, a::~a, base2::~base2* и, наконец, *base1::~base1*.

При вызове конструктора подкласса данные, подлежащие передаче в конструкторы его базовых классов и конструкторы его данных-членов, указываются в списке инициализации конструктора подкласса. Этот список дается после списка аргументов функции-конструктора подкласса, но до определения тела функции:

```
<подкласс>::<подкласс> ( <список_аргументов> ) : <список инициализации>
{
   /* тело */
}
```

<список инициализации> можно представить так:

```
<имя_класса> ( <аргумент> ) [, <имя_класса> ( <аргумент> ) ]+
```

а функцию-конструктор класса *sub* можно записать следующим образом:


```
sub::sub( int x, int y, int z ) : base1(a), base2(b), a(z), b(z=1)
{
   /* тело функции */
}
```

Список инициализации приводится только в определении функции-конструктора подкласса, но не при ее объявлении. Более того, если конструктор класса не определен или определен, но не имеет аргументов, то имя этого базового класса или имя данного-члена в списке инициализации можно не указывать.

2.6. Виртуальные функции

Функция-член класса может объявляться ключевым словом virtual. Это означает, что все подклассы данного класса имеют право переопределять эту виртуальную функцию. Так, в C++ поддерживается полиморфизм — одна из основных особенностей объектно-ориентированного программирования. Подкласс может переопределять или не переопределять виртуальные функции, которые наследует от своих базовых классов. В первом случае он, однако, не имеет права изменять сигнатуру переопределяемых функций.

Виртуальные функции используются при определении общих операций для совокупности родственных классов. Интерфейс этих операций одинаков для всех классов, чего нельзя сказать о реальном поведении (и реализации) этих операций в каждом классе. Например, базовый класс *menu* может определить операцию *draw*, предназначенную для изображения на экране какой-то фигуры, а его подкласс *window* может переопределить эту операцию, которая после этого будет представлять на экране окно, а затем меню.

Функции-конструкторы в качестве виртуальных объявлять нельзя, а вот функции-деструкторы и перегруженные функции операций можно и нужно объявлять как виртуальные.

Пример использования виртуальных функций приведен ниже:

```
// source module: virtual.C
#include <iostream.h>
class date
{
      int year, month, day;
   public:
      date(int y, int m, int d)   { year=y; month=m; day=d; );
      virtual ~date() {};
      virtual void print() {cerr << year << '/' << month << '/'
                        <<day << "\n";);
      virtual void set (int a, int b, int c){ year=a;  month=b; day=c; };
};
class derived : public date
{
      int x;
    public:
      derived (int a,int b,int c,int d): date(a,b,c), x(d) {};
```

```
    ~derived() {};
    void print() { date::print(); cout << "derived: x=" << x <<"\n"; };
    virtual void set(int a, int b, int c) {x=a;};
};
int main()
{
    date foo(1997,5,4);
    derived y(1,2,3,4);
    date* p = &y;
    p->print();        // derived::print()
    p = & foo;
    p->print();        // date::print()
    return 0;
}
```

В этом примере функции *date::~date, date::print* и *date::set* объявлены как виртуальные. Класс *derived* переопределяет виртуальные функции *print* и *set*. Когда в функции *main* переменная *p* указывает на *y*, оператор *p->print()* фактически вызывает функцию *derived::print*, а когда *p* указывает на *foo*, оператор *p->print()* вызывает функцию *date::print*. Если бы функция *date::print* не была объявлена как виртуальная, то оба оператора в функции *main* вызывали бы только функцию *date::print*, а функция *derived::print* трактовалась бы как перегруженная функция функции *date::print*.

Компиляция и пробный запуск этой программы дают следующие результаты:

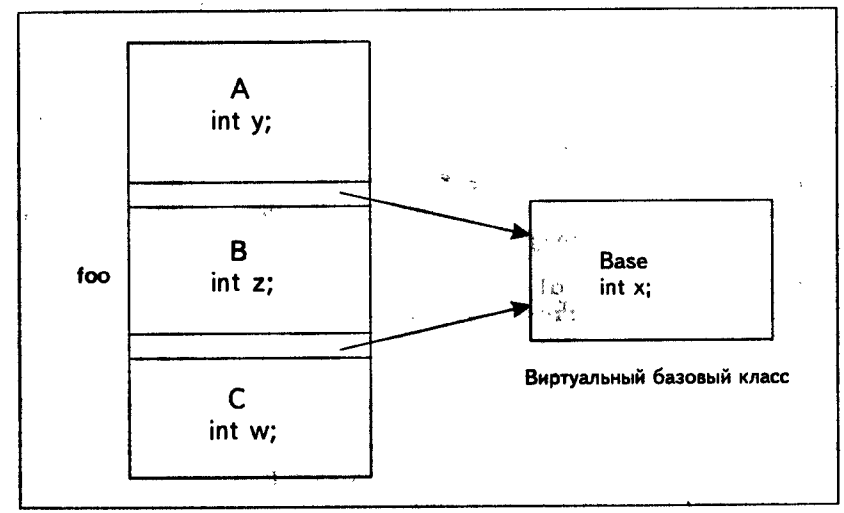
```
% CC virtual.C
% a.out
1/2/3
derived: x=4
1997/5/4
```

2.7. Виртуальные базовые классы

Допустим, класс *A* и класс *B* порождены от класса *Base*. Если класс *C* произвести и от класса *A*, и от класса *B*, то у каждого объекта класса *C* будет по два экземпляра данных-членов класса *Base*, что, возможно, для какого-то приложения и нежелательно. Дабы гарантировать, что в такой ситуации (с множественным наследованием) каждому объекту класса *C* будет доступен только один экземпляр данных-членов класса *Base*, необходимо объявить класс *Base* как виртуальный и в объявлении класса *A*, и в объявлении класса *B*:

```
class Base
{
   int x;
   ...
};

class A: virtual public Base
{
   int y;
   ...
};
class B: virtual public Base
{
   int z;
   ...
};

class C: public A, private B
{
   int w;
   ...
};
```

Схема хранения объекта (например, *foo*) класса C будет выглядеть примерно так:



Виртуальный базовый класс должен определить функцию-конструктор, которая либо не принимает аргументов, либо использует для всех своих аргументов значения по умолчанию. Виртуальный базовый класс инициализируется своим последним производным классом (например, класс *Base* инициализируется с помощью класса *C*, а не классом *A* или *B*). Если последний производный класс не инициализирует виртуальный базовый класс явно, то вызывается конструктор виртуального базового класса, который не требует аргументов и инициализирует объекты последнего производного класса. Конструкторы виртуальных базовых классов вызываются до конструкторов невиртуальных базовых классов, а их деструкторы — после деструкторов невиртуальных базовых классов.

Если в каком-то подклассе существуют открытый и закрытый виртуальные базовые классы, то виртуальный базовый класс считается открытым в

последнем производном классе. Например, в приведенном выше примере класс *Base* считается открытым базовым классом в классе *C*.

Ниже демонстрируется последовательность вызовов функций-конструкторов и функций-деструкторов виртуальных и невиртуальных базовых классов, а также вызов функции виртуального базового класса (*base::print*) в объекте последнего производного класса:

```
// source module: virtual_base.C
#include <stream.h>

class base {
    public:
      int x;
      base(int xa=0) : x(xa) { cerr << "base(" << xa << ")
                                 called\n"; };
      virtual void print()   { cerr << "x=" << x << "\n"; };
      virtual ~base()        { cerr << "~base() called\n"; }
};

class d1 : virtual public base
{
    public:
      int y;
      d1(int xa, int ya) : base(xa)
      {
           cerr<<"d1("<<xa<<","<<ya<<") called\n";    y = ya;
      };
      ~d1() { cerr << "~d1() called\n"; };
      void print() { cerr << "y=" << y << "\n"; };
};

class d2 : virtual public base
{
    public:
      int z;
      d2(int xa, int za) : base(xa)
      {
           cerr<<"d2("<<xa<<","<<za<<") called\n"; z = za;
      };
      ~d2() { cerr << "~d2() called\n"; };
};

class derived : public d1, public d2
{
    public:
      int all;
      derived(int a, int b, int c, int d) : base(a),
d1(a,b), d2(a,c), all(d)
      { cerr << "derived(" << all << ") called\n"; };
      void prints() { base::print(); cerr << "all=" << all << "\n"; };
};

int main()
{
    derived foo(1,2,3,4);
    foo.prints();
    return 0;
}
```

Компиляция и пробный запуск этой программы дают следующие результаты:

```
% CC virtual_base.C
% a.out
base(1) called
d1(1,2) called
d2(1,3) called
derived(4) called
x=1
all=4
~d2() called
~d1() called
~base() called
```

2.8. Абстрактные классы

Абстрактным называется класс, служащий в качестве базового для других классов. Он содержит неполную спецификацию своих операций, поэтому для абстрактного класса никаких объектов определять не следует. Компилятор C++ отслеживает это ограничение.

В абстрактном классе объявляется одна или несколько чистых (pure) виртуальных функций; эти функции не имеют определений, и все подклассы этого абстрактного класса должны их переопределять. Чистая виртуальная функция объявляется следующим образом:

```
class abstract_base
{
  public:
     virtual void draw() = 0; // чистая виртуальная функция
};
```

Абстрактный класс должен содержать как минимум одно объявление чистой виртуальной функции. Пользователи могут определять переменные-указатели и переменные-ссылки, которые должны ссылаться только на объекты подклассов.

Проиллюстрируем использование абстрактного класса на примере интерактивной программы, выдающей пользователю меню, в котором он должен выбирать необходимые операции. Каждая операция инкапсулируется одним объектом типа *menu_obj*. Класс *menu_obj* произведен от класса *abstract_base*. Последний представляет собой абстрактный базовый класс и содержит две чистых виртуальных функции: *info* и *opr*, которые переопределяются в классе *menu_obj*. Функция *info* вызывается для демонстрации одного

из способов использования объекта класса *menu_obj*, а функция *opr* — для выполнения реальной операции с объектом типа *menu_obj*.

В этом примере определены три объекта класса *menu_obj*: один — для вывода на экран значения местного времени и даты, второй — для вывода значения времени и даты по Гринвичу, а третий — для завершения программы. Эти объекты хранятся в диспетчерской таблице, которая называется *menu*. Функция *main* организовывает циклическое выполнение функции *menu_obj::info* для всех объектов, которые хранятся в *menu*, с целью вывода меню на консоль. Затем она принимает от пользователя входные данные (выбор элемента меню) и вызывает функцию *menu_obj::opr* соответствующего объекта. Отметим, что данная программа предусматривает возможность модернизации. Пользователи могут определять новые объекты класса *menu_obj* и записывать их в *menu*, а программа будет автоматически вводить эти объекты в действие:

```
// source module: abstract.C
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
typedef void (*PF)();

 class abstract_base                    // абстрактный базовый класс
{
   protected:
      PF       fn_ptr;
      char     *info_msg ;             // для отображения информации
   public:
      abstract_base(PF fn=0, char* msg=0) (fn_ptr=fn, info_msg=msg; );
      virtual void info(int) =0;
      virtual void opr() =0;
);

class menu_obj : public abstract_base   // производный класс
{
   public:
      menu_obj(PF fn, char *msg) : abstract_base(fn,msg) ();
      void info(int menu_idx)
      {
         cout << menu_idx << ": " << info_msg << "\n";
      );
      void opr() ( this->fn_ptr(); );
};

inline void fn0()
{
       long tim = time(0);
       cerr<<"Local: "<<asctime(localtime(&tim))<<"\n";
)

inline void fn1()
{
       long tim = time(0);
       cerr<<"GMT: "<<asctime(gmtime(&tim))<<"\n";
)
inline vold fn2() { exit(0); )

// dispatch table
menu_obj menu[] =
(
      menu_obj(fn0, "Local data:time"),
      menu_obj(fn1, "GMT date:time"),
      menu_obj(fn2, "Exit program")
);

#define  MENU_SIZ  sizeof(menu)/sizeof(menu[0])

inline void display_menu()
(
     for (int i=0; i<MENU_SIZ; i++) menu[i].info(i);
)

int main()
(
     for (int idx; 1; )
     (
        display_menu();
        cout << "Select (0-" << (MENU_SIZ-1) << ")> ";
        cin>>idx;
        if (idx >=0 && idx<MENU_SIZ)
          menu(idx].opr();
        else cerr << "Illegal input: " << idx << "\n";
     )
return 0;
)
```

Компиляция и пробное выполнение этой программы дают следующие результаты:

```
% CC abstract.C
% a.out
0: Local date:time
1: GMT date:time
2: Exit Program
Select (0-2): 0
Local: Sun Apr 13 19:38:21 1997

0: Local date:time
1: GMT date:time
2: Exit Program
Select (0-2): 1
GMT: Sat Apr 12 02:38:22 1997

0: Local date:time
1: GMT date:time
2: Exit Program
Select (0-2): 2
```

2.9. Операции new и delete

В C++ определены операции *new* и *delete*, предназначенные для динамического управления памятью. Считается, что их применение с этой целью более эффективно, чем использование C-функций *malloc*, *calloc* и *free*.

Аргументами операции *new* являются тип данных (или класс) и, при необходимости, заключенный в круглые скобки список данных инициализации для функции-конструктора нового объекта. Если данные инициализации не указаны, то либо вызывается функция-конструктор нового объекта, которая не принимает аргументов (если она определена), либо новый объект не инициализируется.

Пусть, например, есть такое объявление класса:

```
class date {
      int year, month, day;
    public:
      date( int a, int b, int c ) { year=a, month=b, day=c; };
      date() {year = month = day = 0; };
      ~date() {};
};
```

В представленных ниже двух операторах используются два разных конструктора класса *date*:

```
date *date1p = new date (1997,7,1); // используется date::date(int,int,int);
date *date2p = new date;            // используется date::date();
```

С помощью операции *new* можно выделить массив объектов. Для этого нужно указать имя класса и (в квадратных скобках) число объектов в массиве. Так, следующий оператор выделяет массив из десяти объектов, тип которых задает класс *date*:

```
date  *dateList = new date [ 10 ];
```

Чтобы инициализировать объекты массива, выделяемые операцией *new*, класс этого объекта должен иметь конструктор, не требующий аргументов и используемый для инициализации каждого объекта в массиве. Если такой конструктор не определен, объекты массива не инициализируются.

В стандартной библиотеке C++ определена глобальная переменная *_new_handler*. Если она является указателем на заданную пользователем функцию, то каждый раз, когда операция *new* не может быть выполнена, вызывается эта подпрограмма, выполняющая определяемые пользователем действия по устранению ошибки. Затем подпрограмма возвращает вызвавшей ее программе значение указателя NULL. Если *_new_handle* установлена в свое стандартное значение NULL, то при сбое операция *new* просто возвращает вызвавшей ее программе значение указателя NULL.

Переменная *_new_handler* объявляется в заголовке <new.h> следующим образом:

```
extern void (*_new_handler)();
```


Этой переменной можно задавать значение либо прямым присваиванием в пользовательских программах, либо с помощью макрокоманды *set_new_handler* из <new.h>:

```
#include <new.h>
extern void error_handler();  // определяемая пользователем функция
main()
{
      _new_handler = error_handler;        // прямое присваивание
      set_new_handler ( error_handler );   // присваивание через макрос
}
```

Наконец, операции *new* можно дать указание задействовать заранее выделенную область памяти для размещения в ней динамических объектов. В этом случае пользователь принимает на себя задачу по распределению памяти, а операция *new* используется для инициализации новых объектов, помещаемых в указанную им область памяти. Ниже показано, как это делается:

```
#include <new.h>
#include "date.h"

const NUM_OBJ = 1000;
date *pool = new char[size(DATE)*NUM_OBJ];
int main()
{
      date *p = new (pool) date [NUM_OBJ];
      delete [NUM_OBJ] p;
}
```

В этом примере пользователь выделяет область памяти, на которую указывает переменная *pool*, а затем выделяет память под NUM_OBJ объектов класса *date*. Массив данных размещается в области памяти, начинающейся с адреса *pool*, и на него указывает переменная *p*. С помощью операции *delete* он освобождается.

Динамический объект, выделенный операцией *new*, всегда освобождается операцией *delete*. Например, чтобы удалить объект класса *date*, на адрес которого указывает переменная *p*, нужно сделать следующее:

```
date *p = new date;
...
delete p;
```

Если подлежащий удалению объект — это массив, то после ключевого слова *delete* необходимо указать число элементов массива, а затем его имя. Например:

```
date *arrayP = new date[10];
...
delete [10] arrayP;
```

Применение такого синтаксиса обеспечивает вызов функции-деструктора объектов массива для каждого из них. Если этот массив освобождается операцией

```
delete arrayP;
```

то функция-деструктор вызывается только для первого объекта массива.

Операции *new* и *delete* могут перегружаться в классе. В этом случае при выделении объекта такого класса с помощью операции *new* и освобождении его посредством операции *delete* используется экземпляр определенных в данном классе операций.

Перегруженные операции *new* и *delete* должны объявляться как функции-члены класса. Они рассматриваются как статические функции-члены, не имеющие возможности изменять данные-члены объектов своих классов.

В следующем примере объявляются перегруженные операции *new* и *delete* в классе *date*:

```
class date
{
     int year, month, day;
    public:
       date(int a, int b, int c) { year=a,month=b, day=c; };

       ~date() {};

       // перегруженная операция new
       void* operator new (size_t_siz )
       {
          return::new_char[siz];
       };
       // перегруженная операция delete
       void operator delete (void* ptr)
       {
         ::delete ptr;
      };
};
```

Перегруженная функция-член *new* должна принимать аргумент типа *size_t*, который задает размер выделяемого объекта в байтах. Эта функция возвращает затем адрес вновь выделенного объекта.

Перегруженная функция-член *delete* должна принимать аргумент *void**, который указывает на освобождаемый объект. Эта функция не возвращает никакого значения.

Пользователи могут реализовать тела функций-членов *new* и *delete* так, как считают нужным.

2.10. Перегрузка операций

C++ позволяет пользователям определять стандартные встроенные операции для работы с классами. Это дает возможность использовать объекты классов так, как будто они содержат данные, относящиеся к базовому типу. Встроенные операции "+", "-", "*" и "[]" для объектов классов смысла не имеют, если только пользователи не перегрузят их явно в своих классах.

+	-	*	/	%	^	&	|
~	!	,	=	<	>	<=	>=
++	--	<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=	|=	<<=
>>=	[]	()	->	->*	new	delete	

В таблице показаны знаки операций C++, которые можно перегружать в классах. При этом значение, очередность выполнения, число операндов перегрузкой не отменяются и, следовательно, операции "+", "-", "*" и "&" могут определяться как унарные или бинарные. Более того, при перегрузке все префиксные и постфиксные операции "++" и "--" рассматриваются как префиксные.

Все перегруженные функции операций должны принимать в качестве аргумента минимум один объект класса. Они могут объявляться как дружественные функции или как функции-члены. При этом операции "<<" и ">>", которые не требуют, чтобы левый операнд был объектом класса, должны определяться как дружественные функции, а операции "[]", "=", "->", "()" и "+=", которые требуют, чтобы левый операнд был объектом класса,— как функции-члены.

Перегрузка операций демонстрируется в следующем примере:

```
// source module: overload.C
#include <iostream.h>
#include <string.h>
class sarray
{
     int num;
     char *data;
```

```
public:
  sarray( const char* str ) // конструктор
  {
    num = strlen(str)+1;
    data = new char[num];
    strcpy( data, str );
  };
  ~sarray() { delete data; };

  char& operator[] {int idx ) // деструктор
  {
    if (idx>=0 && idx<num)
        return data(idx];
    else
     {
        cerr << "Invalid index: " << idx << endl;
        return data[0];
     );
  );

  const char* operator=( const char* str )
  {
    if (strlen(str))
     {
        delete data;
        num = strlen(str)+1;
        data = new char[num];
        strcpy (data, str );
     }
    return str;
  };

  const sarray& operator=( const sarray& obj )
  {
    if (strlen(obj.data))
    {
        delete data;
        num = strlen(obj.data)+1;
        data = new char[num];
        strcpy (data, obj.data );
    }
    return obj;
  };

  int operator == ( const char* str )
  {
     return (str && data) ?  strcmp(data,str) : -1;
  };

  int operator < ( const char* str)
  {
    return (strcmp(data,str) < 0) ? 1 : 0;
  };
```



```
    int operator > ( const char* str )
    {
      return (strcmp(data,str) > 0) ? 1 : 0;
    };

    friend ostream& operator << (ostream& os, sarray& obj)
    {
       return os << (obj.data ? obj.data : "Nil") ;
    );
);

int main()
{
    sarray A("Hello"), B("world");
    cout << "A: " << A << endl;          // операция <<
    A = "Bad";                           // операция =
    A(0] = 'T';                          // операция (]
    cout << "A: " << A << endl;
    A = B;                               // операция =array&
    cout << "A: " << A(1] << endl;
    cout << "A < B: " << (A < "Two") << endl;
    return 0;
}
```

В этом примере определяется класс *sarray*, который организует массив символов для каждого из своих объектов. Преимущество использования этого класса состоит в том, что каждый из его объектов динамически настраивает свой буфер для хранения любой строки символов. Более того, объектами можно оперировать с помощью операций "<<", "[]", "=" и "<", которые объявлены как перегруженные функции. Все они делают объекты *sarray* более доступными для понимания и использования, избавляют пользователей от необходимости манипулировать массивом на более низком уровне.

Компиляция и пробное выполнение этой программы дают следующие результаты:

```
% CC overload.C
% a.out
A: Hello
A: Tad
A: o
A < B: 0
```

2.11. Шаблоны функций и шаблоны классов

Шаблоны функций и шаблоны классов позволяют пользователям создавать родовые функции и классы, работающие с разными типами данных и классами. После их кодирования и тестирования пользователи могут создавать разные экземпляры этих функций и классов для конкретных типов данных и (или) классов. Таким образом, применение шаблонов функций и классов позволяет добиться существенной экономии времени разработчика.

Еще одно преимущество шаблонов — сокращение (в расчете на одну программу) числа уникальных функций и имен классов, требующих определения. Это ускоряет процесс компиляции и снижает вероятность конфликтов имен в пользовательских программах.

Экземпляр шаблона функции или класса создается при первом использовании или при указании адреса шаблона. Он не имеет имени и уничтожается, как только перестает быть нужным.

2.11.1. Шаблоны функций

Синтаксис объявления шаблона функции представлен ниже:

```
template <список_формальных_параметров> <возвращаемое_значение>
<имя_функции> ( <список_аргументов> )
{
   <тело>
}
```

Шаблон функции, меняющей местами содержимое двух объектов, определяется следующим образом:

```
template <class T> void swap ( T& left, T& right )
{
    T temp = left;
    left = right;
    right = temp;
}
```

После объявления подобной шаблонной функции пользователи могут создавать объекты специализированных экземпляров этого класса для различных типов данных. Покажем, как это делается:

```
main()
{
    int a = 1, b = 2;
    swap(a,b);                // создает экземпляр swap(int,int)
    double aa = 101.0, bb = 25.0;
    swap(aa,bb);              // создает экземпляр swap(double,double)
}
```

Список формальных параметров заключается в символы < и >. Он содержит формальные параметры типа, отделенные друг от друга запятыми. Этот список не может быть пустым. Каждый формальный параметр начинается с ключевого слова *class*, а далее следует идентификатор типа. Например, правильным будет такое объявление:

```
template <class T, class U> void foo(T*, U);
```

Следующее определение — неверное, поскольку перед идентификатором типа *U* нет ключевого слова *class*:

```
template <class T, U> void fool(T&); // ошибка. Должно быть class U
```


Параметр может встречаться в списке только один раз. Представленное далее объявление содержит ошибку, поскольку задает *class T* дважды:

```
template <class T, class T> void fool(T&); // ошибка
```

Каждый параметр должен быть указан в списке аргументов шаблона функции хотя бы один раз. Следующее объявление ошибочно, потому что в списке аргументов функции отсутствует идентификатор типа *U*:

```
template <class T, class U> U fool(T&); // ошибка
```

Правильным будет использовать идентификатор типа *U* в списке аргументов функции:

```
template <class T, class U> U fool(T&,U*); // нормально
```

Шаблон функции может объявляться как внешний (*extern*), статический (*static*) или встроенный (*inline*). Этот спецификатор ставится после списка параметров, перед спецификацией типа возвращаемого значения. Ниже объявляются встроенная и внешняя функции:

```
template <class T> inline void foo(T* tobj, int size) {...}
template <class T> extern int fooA(T& tobj);
```

Шаблон функции можно перегружать при условии, что в сигнатуре объявлений будут указаны разный тип или различное количество аргументов. Все приведенные ниже объявления допустимы:

```
template <class T> T min(T t1, T t2);
template <class T> T min(T* t1, T t2, T t3);
template <class T> T min(T t1, int t2);
```

Однако следующие два объявления являются неверными, потому что идентификаторы типа параметров нельзя использовать для различения перегруженных шаблонов функции:

```
template <class T> T min(T t1, T t2);
template <class U> U min(U t1, U t2);     // ошибка!
```

И, наконец, можно определить специализированные шаблонные функции. При установлении порядка вызова функций специализированные функции имеют более высокий приоритет, чем родовые шаблоны функций. Например, в следующей программе в первом вызове *min* в функции *main* используется специализированная версия *min*, которая принимает аргументы типа *char**, а второй вызов *min* создает экземпляр шаблона функции *min* для данных типа double:

```
template <class T> T min(T t1, T t2)
{
    return (t1 < t2) ? t1 : t2;
}
    // специализированная версия min()
char* min(char* t1, char* t2)
{
    return (strcmp(t1,t2) < 0) ? t1 : t2;
```

```
}
int main()
{
   char* ptr = min("C++","UNIX");    // min(char*, char*)
   double x = min(2.0, 3.0);         // min(T,T)
}
```

2.11.2. Шаблоны классов

Объявление шаблона класса имеет такой формальный синтаксис:

```
template <список_формальных_параметров> class <имя_класса>
{
   <объявление>
}
```

За ключевым словом *template* в объявлении шаблона класса следует список формальных параметров, затем имя класса и его тело. Список формальных параметров заключается в символы < и >, и параметры в нем отделяются друг от друга запятыми. Вот пример объявления шаблона класса:

```
template <class T, int len> class foo
{
   T list[len];
   ...
};
```

Как и в объявлении шаблона функции, каждый параметр, объявленный в списке формальных параметров, должен быть использован в объявлении соответствующего шаблона класса хотя бы один раз. Более того, при каждой ссылке на имя шаблона класса должен указываться и список параметров, заключенный в угловые скобки (кроме случая, когда это делается внутри объявления класса). Объявим шаблон класса *Array*, который содержит массив типа *T* с количеством элементов *len*. Параметры *T* и *len* должны указываться при создании экземпляров этого шаблона. Обратите внимание на то, что в определении функции-конструктора указаны имя класса и список параметров:

```
template <class T, int len> class Array
{
   public:
      Array();
      ~Array() {};
   protected:
      T list[len];
};

template <class TT, int len> inline
Array<TT,len>::Array()
{
   ....
}
```


Чтобы создать объект для специализированного экземпляра шаблона класса, пользователи указывают имя класса и фактические данные для списка параметров, заключенного в символы < и >. Так, чтобы создать объект класса *Array*, который содержит целочисленный массив из 100 элементов, нужно дать следующее его определение:

```
Array<int, 100> foo;    // foo — это объект
```

Шаблоны классов могут быть порождены от шаблонных и нешаблонных базовых классов. Соотношение типов и подтипов между порожденными и базовыми открытыми шаблонами классов сохраняется при условии, что они имеют фактические параметры одного типа.

Объявим шаблон подкласса *Array_super*, порожденный от шаблонов классов *b1* и *b2*. Переменная *foo* определяется как экземпляр класса *Array_super*, а *ptr* — это указатель на экземпляр класса *b1*. Поскольку экземпляры шаблонов *b1<int>* и *Array_super<int>* имеют параметры одного типа, то они совместимы и, следовательно, указателю *ptr* можно присвоить адрес *foo*. Однако экземпляры шаблонов *b1<double>* и *Array_super<int>* несовместимы, поэтому присваивание адреса *foo* указателю *ptr2* будет ошибкой.

```
template <class Type>
   class Array_super: public b1<Type>, public b2<Type> {....};
Array_super<int> foo;
b1<int> *ptr = &foo;          // правильно
b1<double> *ptr2 = &foo;      // ошибка
```

Внутри шаблонов классов могут объявляться дружественные функции и классы. Каждая из функций может быть:

- функцией или классом, объявленными без шаблона;
- шаблоном;
- специализированным экземпляром шаблона функции.

Сказанное относится и к дружественному классу, объявляемому внутри шаблона.

Если дружественная функция и дружественный класс представляют собой шаблоны, то все экземпляры данной функции или данного класса являются дружественными для шаблонов, внутри которых они объявлены. С другой стороны, если дружественная функция (или класс) является специализированным экземпляром шаблона, то дружественным для объявляемого шаблона будет только этот экземпляр. Данные понятия можно проиллюстрировать следующим образом:

```
template <class U> class Container
{
   // общий шаблон дружественного класса
   template <class T> friend class general_class;

   // общий шаблон дружественной функции
   template <class UT> friend general_func ( <UT>&, int );...
```

```
    // дружественным является только экземпляр этого же типа U класса Array
    friend class Array<U>;

    // дружественным является только экземпляр этого же типа U функции
    friend ostream& operator<< (osteram&, Container<U>&);

    // нешаблонный дружественный класс
    friend class dates;

    // нешаблонная дружественная функция
    frien void foo ();
};
```

В этом примере шаблон класса *Container* имеет формальный параметр типа *U*. Класс *Container* имеет несколько дружественных функций и классов; из них классы *dates* и *foo* являются дружественными для всех специализированных экземпляров класса *Container*. Шаблон класса *Array* и перегруженная операция "<<" являются дружественными для специализированных экземпляров класса *Container*, если они используют параметры одного типа. Таким образом, *Array<int>* — дружественный класс для *Container<int>*, но *Array<double>* таковым не является. Наконец, все специализированные экземпляры шаблона класса *general_class* и шаблона функции *general_func* — дружественные для всех специализированных экземпляров класса *Container*.

Для шаблона класса могут определяться специализированные функции-члены и шаблоны классов. Однако все специализированные экземпляры можно определять только после объявления шаблона класса. Кроме того, в специализированном шаблоне класса должны быть определены все функции-члены шаблона того класса, на котором он построен. В представленном ниже примере объявляются шаблон класса *Array*, затем специализированный конструктор класса *Array* для типа данных *double* и, наконец, определен специализированный класс *Array* типа *char**:

```
template <class T> class Array
{
   public:
      Array(int sz) { ar=new T[size=sz]; };
      ~Array() { delete [sz] ar;);
      T& operator[](int) { return ar(i]; };
   protected:
      T* ar;
      int size;
);

// специализированный конструктор типа double
Array<double>::Array( int size ) { ... }

// определение специализированного класса Array
class Array<char*>
```



```
{
   public:
      Array(int sz)  { ar=new char[size=sz]; };
      ~Array()     { delete [] ar; );
      char& operator[](int i) { return ar(i]; };
   protected:
      char* ar;
      int size;
};
```

В шаблоне класса могут объявляться статические данные-члены. В каждом специализированном экземпляре такого шаблона имеется собственный набор статических данных-членов. Ниже объявляется шаблон класса *Array* с двумя статическими данными-членами: *Array<T>::pool* и *Array<T>::pool_sz*. Эти статические переменные по умолчанию инициализируются соответственно в 0 и 100 для всех специализированных экземпляров класса *Array*:

```
template <class T> class Array
{
   public:
      Array(int sz)  { ar = new char[size=sz]; };
      ~Array()     {delete [sz] ar; );
      void *operator new(size_t);
      void operator delete(void*, size_t);
   protected:
      char*            ar;
      int              size;
      static Array*    pool;
      static const int pool_sz;
);
template <class T> Array<T>* Array<T>::pool = 0;
template <class T> const int Array<T>::pool_sz = 100;
```

Можно определить специализированный экземпляр статических данных-членов класса и задать для них уникальные начальные значения. Так, переменные *Array<char>::pool* и *Array<char>::pool_sz* для экземпляра *char* класса *Array* определяются следующим образом:

```
Array<char>* Array<char>::pool = new char[1000];
Array<char>* Array<char>::pool_sz = 1000;
```

Следует также сказать, что доступ к статическим данным-членам шаблона класса может осуществляться только через специализированный экземпляр этого класса. Так, из представленных ниже трех операторов первый не верен, поскольку обращается непосредственно к *Array<T>::pool*, а обращения к *Array<char>::pool* и *Array<int>::pool_sz* являются корректными:

```
cout << Array<T>::pool << endl;          // ошибка
Array<char>* ptr = Array<char>::pool;    // правильно
int x = Array<int>::pool_sz;             // правильно
```

Нешаблонная функция может манипулировать объектами специализированных экземпляров шаблонов классов, тогда как шаблонная функция может использовать объекты либо конкретного экземпляра, либо общего шаблона класса. В приведенном ниже примере *foo* — нешаблонная функция; следовательно, она может работать с объектами специализированного экземпляра (в данном случае *Array<int>*) класса *Array*. Шаблонная функция *foo2* может манипулировать объектами любого экземпляра класса *Array*, если они имеют параметры одного типа (т.е. функция *foo2<int>* может в качестве аргумента принимать объект типа *Array<int>*):

```
void foo(Array<int>& Aobj, int size )
{
  Array<int> *ptr = &Aobj;
  ....
}

template <class T> extern void foo2 ( Array<T>&, Array<int>& );
```

2.12. Обработка исключительных ситуаций

В ANSI/ISO C++ есть стандартный метод обработки исключительных ситуаций, с помощью которого все приложения реагируют на отклонения, возникающие в ходе выполнения программ. Наличие такого метода упрощает работу по созданию приложений и обеспечивает согласованность в их функционировании.

Исключительная ситуация — это состояние ошибки, обнаруженное в программе в ходе ее выполнения. Исключительная ситуация "генерируется" при помощи оператора *throw*, а функция-обработчик исключительных ситуаций "перехватывается" определяемым пользователем *блоком-ловушкой*, который находится в этой же программе. Если блок-ловушка не прекращает выполнение программы, то управление передается в позицию, находящуюся сразу же за блоком-ловушкой, а не за оператором *throw*. Кроме того, исключительная ситуация может генерироваться только программным кодом, выполняемым прямо или косвенно в блоке *try*. Таким образом, обработка исключительных ситуаций требует специального структурирования пользовательской программы, предусматривающего наличие области кода, где может возникнуть исключительная ситуация, и одного или нескольких блоков-ловушек для ее обработки.

Механизмы обработки исключительных ситуаций в C++ синхронны; генерируются исключительные ситуации в пользовательских приложениях через явные операторы *throw*. Этим они отличаются от асинхронных исключительных ситуаций, генерируемых такими событиями, как нажатие пользователем клавиши на клавиатуре. Асинхронные исключительные ситуации непредсказуемы, т.е. возникают где угодно и когда угодно, и могут обрабатываться с помощью функции *signal* (см. главу 9).

Рассмотрим пример обработки исключительной ситуации в C++:

```
// source module: simple_exception.C
#include <iostream.h>
main( int argc, char* argv[])
{
   try {
      if (argc==1) throw "Insufficient no. of argument";
      while (--argc > 0)
         cout << argc << ": " << argv[argc] << endl;
      cout << "Finish " << argv[0] << endl;
      return 0;
   }
   catch (const char* msg ) {
      cerr << "exception: " << msg << endl;
   }
   cout << "main: continue here after exception\n";
   return 1;
}
```

Здесь нормальный код функции *main* заключен в блок *try*. В этом блоке проверяется значение *argc*. Если оно равно 1, возникает исключительная ситуация, для генерации которой выполняется оператор *throw*. Если же значение *argc* больше 1, то выполняется цикл *while*, в котором в обратном порядке выводятся все аргументы командной строки, и программа завершается через оператор *return 0*.

Оператор *throw* имеет следующий синтаксис:

```
throw <выражение>;
```

где <выражение> — это любое выражение C++, которое при вычислении дает значение одного из базовых типов данных C++ или объект класса. Значение этого выражения используется для выбора блока-ловушки, которому соответствует тип "аргумента" или который совместим с типом данных выражения. Если оператор *throw* выполняется, то остальные операторы в блоке *try* пропускаются и управление передается в выбранный блок-ловушку.

В функции можно задавать один или несколько блоков-ловушек. Они должны указываться сразу после блока *try*. Каждый блок-ловушка начинается с ключевого слова *catch*, за которым следует спецификация типа исключительной ситуации, заключенная в круглые скобки. После спецификации следует один или несколько операторов блока, заключенных в фигурные скобки. Следует отметить, что хотя блок-ловушка выглядит как определение функции, функцией он не является. Это просто набор операторов C++, которые объединены в группу для каждого типа исключительных ситуаций. Исходя из типа исключительной ситуации, оператор *throw* выбирает блок-ловушку для выполнения, а сам тип исключительной ситуации определяется значением *<выражение>* оператора *throw*. Данное значение, как правило, передает дополнительную информацию о генерируемой исключительной ситуации.

Если после выполнения блока-ловушки программа не завершается, управление передается оператору, следующему за ловушками.

В приведенном выше примере компиляция и пробный запуск программы без исключительных ситуаций дали следующие результаты:

```
% CC simple_exception.C
% a.out hello
1: hello
Finish a.out
```

Если перезапустить эту программу без указания какого-либо аргумента, то возникнет исключительная ситуация и программа выдаст такие результаты:

```
% a.out
exception: Insufficient no. of argument
main: continue here after exception
```

В качестве аргумента в операторе *throw* можно указывать и имя объекта класса. Это позволяет оператору *throw* передавать в блок-ловушку больше информации, что расширяет возможности диагностики ошибок и уведомлений о них. Приведенная ниже программа — это переработанный вариант предыдущего примера:

```
// source module: simple2.C
#include <iostream.h>

class errObj {
   public:
      int line;
      char* msg;
      errObj( int lineNo, char* str )
      {
         line = lineNo;
         msg = str;
      };
      ~errObj() {};
};

main( int argc, char* argv[])
{
    try
    {
      if (argc==1) throw errObj(__LINE__,"Insufficient no. of arguments");
      while (--argc > 0)
         cout << argc << ":" << argv[argc] << endl;
      cout << "Finish " << argv[0] << endl;
      return 0;
    }
   catch (errObj& obj )
    {
      cerr << "exception at line: " << obj.line << ", msg: " << obj.msg << endl;
    }
```



```
      cout << "main: continue here after exception\n";
       return 1;
}
```

Компиляция и пробный запуск этой программы с исключительной ситуацией дали следующие результаты:

```
% CC simple2.C
% a.out
exception at line: 23, msg: Insufficient no. of arguments
main: continue here after exception
```

Если исключительная ситуация генерирована, но ни один из блоков-ловушек в этой же функции не соответствует аргументу оператора *throw*, то функция "возвращается" в вызвавшую ее функцию (или функции) и на соответствие аргументу оператора *throw* проверяются блоки-ловушки в каждой из этих функций. Данный процесс прекращается в том случае, если обнаружен соответствующий блок-ловушка и выполнение программы продолжается в этом блоке и следующем за ним фрагменте кода либо если соответствия нет и вызывается встроенная функция *terminate*. Функция *terminate*, в свою очередь, вызывает функцию *abort*, которая прерывает программу.

Когда функция возвращается в вызвавшую ее функцию (или функции) вследствие выполнения оператора *throw*, все объекты, локальные для завершившей работу функции, освобождаются через свои деструкторы, и область динамического стека, зарезервированного для этой функции, также освобождается.

Эту концепцию иллюстрирует следующий пример:

```
// source module: simple3.C
#include <iostream.h>
void f2 ( int x )
{
   try {
       switch (x) {
         case 1: throw "exception from f2.";
         case 2: throw 2;
       }
       cout << "f2: got " << x << " arguments.\n";
       return;
     }
   catch (int no_arg ) {
       cerr << "f2 error: need at least " << no_arg << " arguments\n";
   }
   cerr << "f2 returns after an exception\n";
}

main( int argc, char* argv[])
{
    try {
       f2(argc);
```

```
    cout << "main: f2 returns normally\n";
    return 0;
  }
  catch (const char* str ) {
    cerr << "main: " << str << endl;
  }
  cerr << "main: f2 returns via an exeception\n";
   return 1;
}
```

Если программа вызывается без аргументов, то функция *f2* выполняет оператор *throw "exception from f2"*, что инициирует выполнение блока *catch (const char* str)* в функции *main*. Программа дает следующие результаты:

```
% CC simple3.C
% a.out
main: exception from f2
main: f2 returns via an exeception
```

Если же программа вызывается с одним аргументом (и значение *argv* равно 2), то в функции *f2* выполняется оператор *throw 2*, что инициирует выполнение блока *catch (int no_arg)* в функции *f2*. Результаты программы:

```
% a.out hello
f2 error: need at least 2 arguments
f2 returns after an exception
main: f2 returns normally
```

Наконец, если программа вызывается с двумя и более аргументами, то *f2* не создает исключительной ситуации, а результаты выполнения программы будут таковыми:

```
% a.out hello world
f2: got 3 arguments.
main: f2 returns normally
```

2.12.1. Исключительные ситуации и соответствие им блоков-ловушек

Когда выполняется оператор *throw* с аргументом типа *T*, то при выборе блока-ловушки для перехвата исключительной ситуации руководствуются определенными правилами. В частности, если тип данных блока-ловушки — *C*, то блок-ловушка выполняется при соблюдении любого из следующих условий:

- Тип *T* совпадает с типом *C*;
- *T* — это тот же тип данных, что и *C*, но один из них определен с помощью ключевого слова *const* или *volatile*;
- *C* является ссылкой на *T* или наоборот;
- *C* — это открытый или защищенный базовый класс для *T*;

- *T* и *C* — указатели. *T* можно преобразовать в *C* путем стандартного преобразования указателей.

2.12.2. Объявление функций с оператором throw

В объявлении функции при необходимости можно задать набор исключительных ситуаций, которые она будет прямо или косвенно генерировать. Этот набор вводится в виде *списка генерации* (throw list). Например, следующий оператор объявляет внешнюю функцию *funct*, которая может генерировать исключительные ситуации с данными типа *const char** или *int*.

```
extern void funct (char* ar) throw(const char*, int);
```

Список генерации может быть пустым. Это значит, что функция не будет генерировать никаких исключительных ситуаций. Приведенный ниже оператор объявляет функцию *funct2*, которая не будет возбуждать никаких исключительных ситуаций ни прямо, ни косвенно:

```
extern void funct2(char* ar) throw();
```

Если же функция *funct2* все-таки вызовет оператор *throw*, то будет вызвана встроенная функция *unexpected*. По умолчанию эта функция вызывает функцию *abort*, которая прерывает программу.

Список генерации не является частью объявления функции, следовательно, он не может использоваться для перегрузки функций. Например, компилятор C++ считает следующие два объявления функций одинаковыми:

```
extern void funct3(char* ar) throw(char*);
extern void funct3(char* ar=0);
```

2.12.3. Функции terminate и unexpected

Функция *terminate* вызывается в том случае, если оператор *throw* выполняется, но соответствующий блок-ловушка не обнаруживается. По умолчанию эта функция вызывает функцию *abort*, которая прерывает программу. С помощью функции *set_terminate* пользователи могут инсталлировать в *terminate* вместо *abort* свои функции:

```
extern void user_terminate( void );
void (*old_handler)(void);
old_handler = set_terminate( user_reminate );
```

В этом примере *user_terminate* — определенная пользователем функция, которая должна вызываться при активизации *terminate*. Функция *user_terminate* инсталлируется посредством функции *set_terminate*, а старая функция, инсталлированная в функции *terminate*, сохраняется в переменной *old_handler*.

Функция *unexpected* вызывается в том случае, если оператор *throw* выполняется в функции, которая объявлена с пустым списком генерации. По умолчанию функция *unexpected* вызывает функцию *terminate*, которая прерывает

работу программы. С помощью функции *set_unexpected* пользователи могут инсталлировать в *unexpected* вместо *terminate* свои функции:

```
extern void user_unexpected( void );
void (*old_handler)(void);
old_handler = set_unexpected( user_unexpected );
```

В данном примере *user_unexpected* — определенная пользователем функция, которая должна вызываться при активизации функции *unexpected*. Функция *user_unexpected* инсталлируется посредством функции *set_unexpected*, а старая функция, инсталлированная в функции *unexpected*, сохраняется в переменной *old_handler*.

Поскольку предполагается, что функции *terminate* и *unexpected* никогда не выполняют возврат в вызывающие их функции, то все инсталлируемые пользователем функции, которые будут вызываться функциями *terminate* или *unexpected*, заканчиваясь, должны завершать и саму программу.

В хорошо продуманной программе функции *terminate* и *unexpected* должны вызываться редко, поскольку их применение говорит о том, что в пользовательских программах не учтены все возможные исключительные ситуации, т.е. о недостаточно высоком уровне программирования. Единственным исключением является использование библиотек C++ третьих фирм, когда генерируются не указанные в документации исключительные ситуации. В этом случае пользователи должны сообщать о возникших проблемах своим поставщикам и инсталлировать программы-обработчики функций *terminate* и (или) *unexpected* лишь в качестве временной меры.

2.13. Заключение

В этой главе рассматриваются принципы объектно-ориентированного программирования и свойства языка C++, соответствующего проекту стандарта ANSI/ISO. Отмечается, что C++ построен на базе языка C с добавлением конструкций, поддерживающих объектно-ориентированную технологию программирования. К таким конструкциям относятся объявление классов, наследование классов, полиморфизм на базе виртуальных функций, шаблоны функций и классов. Кроме того, рассматривается принятый в C++ метод обработки исключительных ситуаций. Практическое использование новых конструкций иллюстрируется на обширных примерах.

Изложенный здесь материал поможет пользователю систематизировать имеющиеся знания о методах программирования на языке C++, разобраться в информации, представленной в остальных главах книги. Возможно, некоторые читатели и не знакомы с новыми возможностями ANSI/ISO C++, которые описаны в этой главе достаточно подробно.

Глава 3 дает представление о библиотеках потоков ввода-вывода C++, которые интенсивно используются всеми приложениями, написанными на C++, поможет читателям изучить некоторые особенности их применения.

2.14. Литература

1. Andrew Koenig, *Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++ (Committees: WG21/NO414, X3J16/94-0025)*, January 1994.
2. Margaret A. Ellis, and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.
3. Bjarne Stroustrup, *The C++ Programming Language*, Addison Wesley, 1991.

ГЛАВА 3

Классы потоков ввода-вывода языка C++

В этой главе рассматриваются определенные в проекте стандарта C++ ANSI/ISO классы потока ввода-вывода в языке C++. С их помощью пользователи выполняют операции ввода-вывода со стандартными потоками ввода и вывода, файлами и буферами символов. Классы потоков в значительной мере устраняют необходимость использовать функции потокового ввода-вывода языка C, строковые функции и функции класса printf. Преимущество применения классов потока ввода-вывода C++ заключается в том, что они позволяют компилятору в большей степени контролировать типы указываемых фактических параметров, а также могут быть расширены для поддержки классов, определяемых пользователем.

Для классов потока ввода-вывода существуют три основных заголовка. В заголовке <iostream.h> объявляются классы *istream*, *ostream* и *iostream*, предназначенные для проведения операций ввода-вывода со стандартными потоками, а также используемые в большинстве программ на C++ объекты *cout*, *cin*, *cerr* и *clog*. В заголовке <fstream.h> объявляются классы *ifstream*, *ofstream* и *fstream* для операций ввода-вывода с дисковыми файлами, а в заголовке <strstream.h> — классы *istrstream*, *ostrstream* и *strstream* для форматирования данных с использованием буферов символов.

Хотя классы потока ввода-вывода, в особенности такие объекты, как *cout*, *cin* и *cerr*, широко используются в приложениях, написанных на языке C++, не все пользователи в полной мере знакомы с их свойствами; многие учебники по программированию на C++ также не содержат их полных характеристик. Поэтому в настоящей главе дается исчерпывающее описание упомянутых классов.

3.1. Классы ввода-вывода в стандартные потоки

В заголовке <iostream.h> объявляются, как уже упоминалось, три класса стандартных потоков ввода-вывода: *istream*, *ostream* и *iostream*. Класс *istream* служит для ввода данных из потока ввода, класс *ostream* предназначен для вывода данных в поток вывода, а класс *iostream* — для ввода данных и их вывода в поток. Кроме этих классов, в заголовке <iostream.h> также объявляются четыре объекта.

Потоковый объект	Класс, поток ввода-вывода
cin	*istream*, стандартный поток ввода
cout	*ostream*, стандартный поток вывода
cerr	*ostream*, стандартный поток ошибок при выводе без буферизации
clog	*ostream*, стандартным поток ошибок при выводе с буферизацией

Обратите внимание на то, что объекты *cin*, *cout*, *clog* и *cerr* используют дескрипторы файлов, отличные от тех, которые применяются потоками *stdin*, *stdout* и *stderr* в языке С. Однако пользователи могут заставить объекты *cin* и *stdin* использовать один дескриптор, объекты *cout* и *stdout* — другой дескриптор, а объекты *cerr*, *clog* и *stderr* — третий дескриптор. Это делается с помощью статической функции:

```
ios::sync_with_stdio();
```

Указанная функция должна быть вызвана в пользовательской программе до выполнения операций потокового ввода-вывода.

3.1.1. Класс istream

Класс *istream* предназначен для извлечения данных из потока ввода. К этому классу принадлежит объект *cin*. Ниже перечислены операции пользовательского уровня, определенные для класса *istream*.

Операция	Функция
>>	Извлекает из потока ввода данные любого стандартного типа, разделенные пробельными символами (white space character; к ним относятся пробелы, знаки табуляции, символы новой строки и новой страницы и т.п.)
istream& get(char c), int get()	Извлекает из потока ввода символ. Пробельные символы воспринимаются как допустимые
istream& read(char* buf, int size)	Извлекает из потока ввода *size* байтов данных и помещает их в *buf*

Операция	Функция
istream& getline(char* buf,int limit, char delimiter='\n')	Извлекает из потока данные до тех пор, пока не встретится символ-разделитель или признак конца файла ввода, но не более *limit-1* байтов данных. Извлеченные данные помещаются в *buf*. Символ-разделитель, если таковой обнаруживается, в *buf* не заносится
int gcount()	Возвращает количество байтов, извлеченных последним вызовом *read* или *getline*
istream& putback(char c)	Помещает указанный символ *c* обратно в поток ввода
int peek()	Возвращает значение следующего символа в потоке ввода, но не извлекает его
istream& ignore(int limit=1, int delimiter=EOF)	Отбрасывает до *limit* символов в потоке ввода либо пока не встретится символ-разделитель или признак конца файла (EOF)
streampos tellg()	Возвращает текущее смещение в байтах маркера потока относительно начала потока
istream& seekg(streampos offset, seek_dir d=ios::beg)	Перемещает маркер потока на *offset* байтов от начала файла (если *d*=ios::beg), от текущей позиции маркера (если *d*=ios::cur) или от конца файла (если *d*=ios::end)

Ниже приводится программа, похожая на UNIX-программу *wc*, и показано, как используются операции класса *istream*. Программа подсчитывает количество строк, слов и знаков в стандартном потоке ввода:

```
/* source module: wc.C */
#include <iostream.h>
#include <ctype.h>

int main()
{
  int ch, lineno=0, charno = 0, wordno = 0;
  for (int last=0; cin && (ch = cin.get()) != EOF; last=ch)
    switch (ch)
    {
        case '\n':      lineno++, wordno++;
                        break;
        case '/': if (cin.peek()=='/') {//don't count comments
                        cin.ignore(10000,'\n');
                        lineno++;
                    }
                    else charno++;
                    break;
        default:    charno++;
                    if (isspace(ch) && last!=ch) wordno++;
    }
  cout  << charno << " "  << wordno << " " << lineno << "\n" << flush;
return 0;
}
```

Компиляция и пробное выполнение этой программы дают следующие результаты:

```
% CC wc.C
% a.out < /etc/passwd
557  23  14
```

Операция ">>" может быть перегружена как дружественная функция для каждого класса, определяемого пользователем. Это дает возможность извлекать данные объектов класса точно так же, как это делается с объектами стандартных типов данных C++. Перегруженную функцию ">>" необходимо определить следующим образом:

```
class X        // класс, определяемый пользователем
{   ...
    public:
      friend istream& operator >> (istream& is, X& xObj)
      {
          is >><данные-члены класса X>;
      };
       ...
};
```

3.1.2. Класс ostream

Класс *ostream* используется для направления данных в поток вывода. К этому классу принадлежат объекты *cout*, *cerr* и *clog*. Операции пользовательского уровня, определенные для класса *ostream*, перечислены ниже.

Операция	Функция
<<	Направляет данные любого стандартного типа в поток вывода
ostream& put(char ch)	Помещает символ *ch* в поток вывода
ostream& write(const char*buf, int size)	Помещает *size* байтов данных, находящихся в *buf*, в поток вывода
typedef streampos long; streampos tellp()	Возвращает значение смещения маркера потока (в байтах) относительно начала потока
ostream& seekp(streampos offset, seek_dir d=ios::beg)	Перемещает маркер потока на *offset* байтов от начала файла (если *d*=ios::beg), от текущей позиции маркера (если *d*=ios::cur) или от конца файла (если *d*=ios::end)
ostream& flush()	Принудительно очищает буферы, сбрасывая их содержимое в поток вывода

Приведенные ниже команды демонстрируют использование объектов класса *ostream*:

```
cout << "x=" << x << ",y=" << y << "\n";
cout.put('\n').write("Hello world",11).put('\n');
```


Операция "<<" может быть перегружена как дружественная функция для каждого класса, определяемого пользователем. Это дает возможность пользователям выводить на экран данные объектов класса точно так же, как это делается с экземплярами стандартных типов данных C++. Перегруженную функцию "<<" необходимо определить следующим образом:

```
class X        //класс, определяемый пользователем
{   ...
    public:
      friend ostream& operator << (ostream& os, X& xObj)
      {
         os << <данные-члены класса X>;
         return os;
      };
      ...
};
```

3.1.3. Класс iostream

Класс *iostream* является производным от классов *istream*, *ostream* и обладает всеми их свойствами. Данный класс используется главным образом в качестве базового для класса *fstream*, а последний, как правило,— при определении объектов, предназначенных для чтения и записи файлов.

3.1.4. Класс ios

Классы *istream*, *ostream* и *iostream* содержат виртуальный базовый класс *ios*, который определяет состояние каждого объекта класса потока ввода-вывода. В частности, в классе *ios* объявляются следующие операции:

Операция	Функция
int eof()	Возвращает 1, если в потоке встретился EOF
int bad()	Возвращает 1, если обнаружена недопустимая операция (например, попытка смещения указателя в файле за метку EOF)
int fail()	Возвращает 1, если операция ввода-вывода не выполнена или если *bad()* — истина
int good()	Возвращает 1, если все предыдущие операции ввода-вывода выполнены успешно
int rdstate()	Сообщает статус ошибки потокового ввода-вывода
void clear(bits=0)	Устанавливает битовый вектор состояния ошибки в значение, указанное в *bits*. Если *bits*=0, сбрасывает состояние ошибки в 0
int width(int len)	Определяет ширину поля в *len* при выводе последующих данных или размер буфера в *len-1* при вводе строки символов. Эта подпрограмма возвращает значение предыдущей ширины поля
char fill(char ch)	Устанавливает символ-заполнитель в *ch*. Возвращает предыдущий символ-заполнитель

Операция	Функция
int precision(int)	Устанавливает количество значащих цифр, подлежащих отображению при выводе вещественных чисел. Возвращает предыдущее значение точности представления вещественных чисел
long setf(long bitFlag)	Добавляет бит(ы) формата согласно *bitFlag* к коду, определяющему существующий формат. Возвращает старое значение кода формата. Возможные значения для *bitFlag*: ios::showbase — Отображает основание системы счисления ios::showpoint — Отображает конечную десятичную запятую и нуль ios::showpos — Отображает символ знака (+ или -) для числовых значений ios::uppercase — Использует "X" для отображения шестнадцатеричных чисел (если установлено значение *ios::showbase*) и "E" для представления чисел с плавающей запятой в экспоненциальном формате
long setf(long bitFlag long bitField)	Устанавливает/сбрасывает (в соответствии с *bitFlag*) биты формата, как указано в *bitField*. Возвращает старое значение состояния формата. Возможные значения параметров *bitField* и *bitFlag*: ios::basefield — ios::hex — Устанавливает основание системы счисления в щестнадцатеричное ios::basefield — ios::oct — Устанавливает основание системы счисления в восьмеричное ios::basefield — ios::dec — Устанавливает основание системы счисления в десятичное (по умолчанию) ios::floatfield — ios::fixed — Отображает вещественные числа в десятичной форме ios::floatfield — ios::scientific — Отображает вещественные числа в экспоненциальной форме ios::adjustfield — ios::left — Вставляя символы-заполнители после последнего выведенного значения, выравнивает по левому краю следующий аргумент ios::adjustfield — ios::right — Вставляя символы-заполнители перед выводимым в поток значением, выравнивает по правому краю следующий аргумент ios::adjustfield — ios::internal — Добавляет символы-заполнители после начального знака или указателя системы счисления, но перед выводимым значением ios::skipws — 0 — При вводе пробелы и знаки табуляции не пропускаются ios::skipws — ios::skipws — При вводе пробелы и знаки табуляции пропускаются (по умолчанию)

Все операции ввода-вывода потокового объекта прерываются, если его код состояния ошибки не равен нулю. Пользователи могут проверить состояние ошибки потокового объекта с помощью функции *ios::bad* или *ios::fail*, а также с помощью перегруженной операции "!". Покажем, как это делается:


```
if (!cin || !cout) cerr << "I/O error detected";
if (!(cout << x) || x<0) cout.clear(ios::badbit | cout.rdstate());
if (cout.fail()) clog << "cout fails\n";
```

Состояние ошибки потокового объекта можно сбросить с помощью функции *ios::clear*:

```
if (!cin) cin.clear();
```

Помимо обработки состояний ошибки, класс *ios* используется также для установки параметров форматирования данных потоковых объектов. Причем он обладает такими же широкими возможностями по форматированию данных, как и функции класса *printf* в языке C.

Пример программы демонстрирует использование функций *ios*:

```
// source module: ios.C
#include <iostream.h>
int main()
{
    int x = 1024;
    double y= 200.0;
    static char str[80] = "Hello";
    cout.setf( ios::showbase | ios::showpos | ios::uppercase );
    cout.setf( ios::scientific, ios::floatfield );
    cout.precision(8);
    cout << "'";
    cout.width(10);
    cout.fill('*');
    cout  << x << "', y='" << y << "'\n";
    cout << "'";
    cout.width(7);
    cout.setf(ios::left,ios::adjustfield);
    cout.setf(ios::fixed, ios::floatfield );
    cout << x << "', y='" << y << "'\n";
    cout << "'";
    cout.width(8);
    cout.setf(ios::right,ios::adjustfield);
    cout << str << "'\n";
    return 0;
}
```

В этом примере функции *ios:setf* (*ios::scientific*, *ios::floatfield*) и *ios::precision(8)* устанавливают формат отображения данных с плавающей запятой в экспоненциальном представлении с точностью до восьми знаков. С другой стороны, оператор *ios::setf* (*ios::fixed*, *ios::floatfield*) устанавливает формат отображения чисел с плавающей запятой как чисел с фиксированной запятой. Обратите внимание, что значение однажды установленного формата объекта остается неизменным до тех пор, пока его не отменит следующий вызов *ios::setf*.

Остальная, не описанная здесь часть примера достаточно очевидна. Компиляция и пробное выполнение этой программы дают следующие результаты:

```
% CC ios.C -o ios; ios
'*****+1024', y='+2.00000000E+02'
'+1024**', y='+200.00000000'
'***Hello'
```

3.2. Манипуляторы

Манипулятор — это функция, которую можно включить в операции потокового ввода-вывода для выполнения каких-то особых действий. Например, манипулятор *flush* обычно применяется при работе с объектами класса *ostream* с целью принудительного сбрасывания содержащихся в них буферизованных данных:

```
cout << "A big day" << flush;
```

Простой манипулятор — это функция, которая принимает аргумент *istream&* или *ostream&*, совершает какие-либо операции с объектом и возвращает ссылку на него. Следующий пример демонстрирует определения двух манипуляторов: *tab* и *fld*. Манипулятор *tab* вставляет знак табуляции в поток вывода, а манипулятор *fld* устанавливает для выводимых целочисленных данных восьмеричный формат представления с префиксом О. При этом минимальная ширина поля для отображения значения равна 10:

```
ostream& tab(ostream& os)
{
     return os << '\t';
}

ostream& fld(ostream& os)
{
      os.setf(ios::showbase, ios::showbase);
      os.setf(ios::oct, ios::basefield);
      os.width(10);
      return os;
}
```

Приведенные ниже операторы показывают, как используются манипуляторы:

```
int x = 50, y = 234;
cout << fld << x << tab << y << '\n';
```

В заголовке <iomanip.h> объявляется набор системных манипуляторов, которые обычно используются с объектами потоковых классов. Ниже приведены некоторые из них.

Манипулятор	Функция
flush	Принудительно сбрасывает буферизованные данные в поток вывода
setw(int width)	Устанавливает минимальную ширину поля при выводе следующего значения и максимальный размер буфера (*width-1*) при вводе следующей строки символов
resetiosflags(long bitFlag), setiosflags(long bitFlag)	Сбрасывает или устанавливает указанные биты формата в коде, определяющем существующий формат потока
setprecision(int p)	Устанавливает для следующего подлежащего выводу вещественного числа точность представления, равную *p* знаков после запятой

Ниже показано, как используются некоторых из упомянутых системных манипуляторов:

```
cout << x << setw(5) << y << flush;  // принудительное сбрасывание cout
cin  >> resetiosflags(ios::skipws)   /* пробелы и знаки табуляции не
                                        пропускаются */
     >> c
     >> setiosflags(ios:skipws);     /* пропускать пробелы и знаки
                                        табуляции */
cout << setprecision(8) << Dval;     /* установить указанную точность
                                        представления чисел (в данном
                                        случае 8) */
```

3.3. Классы ввода-вывода файлов

В заголовке <fstream.h> объявляются классы *ifstream*, *ofstream* и *fstream*, предназначенные для манипулирования файлами. По своим функциональным возможностям они аналогичны С-функциям, предназначенным для работы с файлами *fopen*, *fread*, *fwrite*, *fclose* и т.д.

В частности, класс *ifstream* является производным от класса *istream* и дает пользователям возможность доступа к файлам и чтения из них данных. Класс *ofstream*, производный от класса *ostream*, обеспечивает возможность доступа и записи данных в файлы. И, наконец, класс *fstream*, производный от обоих этих классов (*ifstream* и *ostream*), дает пользователям возможность доступа к файлам как для ввода, так и для вывода данных. Конструкторы классов *ifstream* и *ofstream* определяются в заголовке <fstream.h> следующим образом:

```
ifstream::ifstream();
ifstream::ifstream( const char* name, int open_mode=ios::in,
                                int port=filebuf::openprot/* 0644 */);
ofstream::ofstream();
ofstream::ofstream( const char* name, int open_mode=ios::out,
                                int prot = filebuf::openprot);
```

Возможные значения аргумента *open_mode* приведены ниже.

Значение	Смысл
ios::in	Открывает файл для чтения
ios::out	Открывает файл для записи
ios::app	Добавляет новые данные в конец файла. Активизирует режим *ios::out*
ios::ate	После открытия файла указатель перемещается в конец файла. Режим *ios::out* должен устанавливаться непосредственно
ios::nocreate	Возвращает ошибку, если файл не существует
ios::noreplace	Возвращает ошибку, если файл уже существует
ios::trunc	Если файл существует, усекает его предыдущее содержимое. Этот режим подразумевается при открытии файла в режиме *ios::out*, если только последний не установлен совместно с *ios::app* или *ios::ate*

Аргумент *prot* задает права доступа по умолчанию, назначаемые файлу, если он создается функцией-конструктором. Значение по умолчанию *filebuf::openprot* равно 0644. Это значит, что владелец файла может его читать и модифицировать, а остальные пользователи — только читать. Данное значение аргумента не используется, если файл, открываемый конструктором, уже существует.

Как используются классы *ifstream* и *ofstream* показано на следующем примере. Здесь файл с именем *from* открывается для чтения, а другой файл, с именем *to*, открывается для записи. Если оба файла открыты успешно, то содержимое файла *from* копируется в содержимое файла *to*. Если какой-либо из файлов не может быть успешно открыт, то выводится сообщение об ошибке:

```
ifstream source ("from");
ofstream target("to");
if (!source || !target)
   cerr << "Error: File 'from' or 'to' open failed\n";
else for (char c=0; target && source.get(c);)
   target.put(c);
```

Кроме функций-членов, унаследованных от класса *iostream*, классы *ifstream*, *ofstream* и *fstream* определяют также собственные специфические функции.

Функция	Смысл
void open(const char* fname, int mode, int prot=openprot)	Открывает файл и присоединяет его к потоковому объекту
void close()	Отсоединяет файл от потокового объекта и закрывает его
void attach(int fd)	Прикрепляет потоковый объект к файлу, обозначенному дескриптором *fd*
filebuf* rdbuf()	Возвращает указатель на массив *filebuf*, связанный с потоковым объектом

Ниже дана простая программа, которая демонстрирует использование функций *open*, *close* и *attach*, уникальных для классов *fstream*:

```
#include <iostream.h>
#include <fstream.h>
int main(int argc, char *argv[])
{
    ifstream source;
    if (argc == 1 || *argv[1] == '-')
       source.attach(0);     // присоединить к stdin
    else source.open(argv[1], ios::in);
       ...
    if (source.rdbuf() - >is_open)
       source.close()        // закрыть файл, если он открыт
]
```

Наконец, произвольный ввод-вывод файлов можно осуществлять с помощью функций *seekg* и *tellg*, унаследованных классами *fstream* от классов *iostream*. Использование этих функций иллюстрируется на примере следующих операторов:

```
fstream tmp("foo",ios::in|ios::out);
streampos pos = tmp.tellg();   // запомнить положение файла
...
tmp.seekg(pos);                // вернуться к предыдущему положению
...
tmp.seekg(-10, ios::end);      // сместиться на 10 байтов к началу файла
tmp.seekg(0, ios::beg);        // перейти к началу файла
tmp.seekg(20,ios::cur);        // передвинуться на 20 байтов вперед
```

3.4. Классы strstream

В заголовке <strstream.h> объявляются классы *istrstream*, *ostrstream* и *strstream*, предназначенные для форматирования данных, находящихся в оперативной памяти. Эти классы содержат функции, эквивалентные библиотечным функциям *sprintf* и *sscanf* языка C. Преимущество использования этих классов по сравнению с вышеуказанными функциями C заключается в том, что они могут быть перегружены для работы с классами, определяемыми пользователем. Они дают также возможность компилятору C++ контролировать типы фактических параметров программ во время компиляции.

Класс *istrstream* является производным от класса *istream* и дает возможность пользователям извлекать отформатированные данные из буфера символов. Класс *ostrstream* — производный от класса *ostream*. Он позволяет помещать отформатированные данные в буфер символов. Наконец, класс *strstream* является производным от обоих классов, *istrstream* и *ostrstream*. С его помощью можно извлекать и вставлять отформатированные данные, используя буфер символов.

Классы *istrstream*, *ostrstream* и *strstream* не объявляют никаких только им присущих функций-членов. Ниже приводится пример форматирования данных с использованием классов *strstream*:

```
// source module: strstream.C
#include <iostream.h>
#include <strstream.h>
main()
{
    double dval;
    int ival;
    char wd[20];
    static char buf[32] = "45.67 99 Hi";

    // dval= 45.67, ival=99, wd="Hi"
    istrstream(buf) >> dval >> ival >> wd;
    ostrstream(buf,sizeof(buf)) << ival << "<- "<< dval << ','<< wd<< '\0';
    cout << buf << endl;                // "99 <- 45.67,Hi"
    return 0;
}
```

В приведенном выше примере данные (45.67, 99 и Hi) извлекаются из переменной *buf* и присваиваются соответственно переменным *dval*, *ival* и *wd* с помощью класса *istrstream*. Это аналогично использованию функции *sscanf* языка C. В этом примере создается временный объект класса *istrstream*, который использует *buf* в качестве внутреннего буфера для извлечения данных.

Значения переменных *dval*, *ival* и *wd* вновь помещаются в буфер *buf* в другом формате с помощью класса *ostrstream*. Эта операция аналогична использованию функции *sprintf* языка C. Обратите внимание, что оператором создается временный объект класса *ostrstream*, который использует *buf* в качестве внутреннего буфера для вставки данных. Завершающий символ "\0" в операторе необходим для того, чтобы строка, хранящаяся в *buf*, была завершена символом конца строки. Компиляция и пробное выполнение программы дают следующие результаты:

```
% CC strstream.C
% a.out
99<-4 5.67,Hi
```

Если конструктору объекта класса *ostrstream* буфер не указывается, объект создает внутренний динамический массив, предназначенный для хранения входных данных. Пользователь может получить доступ к этому массиву, вызвав функцию *ostrstream::str*. Однако если данная функция вызвана, динамический массив "замораживается". Это значит, что данные посредством объекта *strstream* больше не могут быть вставлены в динамический массив, и пользователю необходимо освободить массив после окончания операции.

Способы применения функции *ostrstream::str* показаны ниже:

```
#include <iostream.h>
#include <fstream.h>
```



```
#include <strstream.h>
int main(int argc, char *argv[])
{
   fstream source;
   if (argc == 1 *argv[1] == '-')
      source.attach(0);      // присоединить к stdin
   else source.open(argv[1],ios::in);

   // читать поток ввода и сохранить во внутреннем массиве
   ostrstream str;
   for (char c=0; str && source.get(c);)
      str.put(c);

   // получить доступ к внутреннему массиву
   char *ptr = str.str();
   // операции с данными массива
   ...

   // освободить массив
   delete ptr;

   // закрыть поток ввода
   source.close();       // закрыть файл
}
```

3.5. Заключение

В этой главе рассматриваются классы потоков ввода-вывода C++. Совместно с определяемыми системой объектами (*cin*, *cout*, *cerr* и *clog*) они широко применяются в большинстве программ на C++, так как позволяют контролировать типы объектов и могут быть расширены для поддержки классов, определяемых пользователем. Поэтому пользователю желательно знать не только их основные, но и более детальные характеристики.

В следующей главе рассматриваются некоторые стандартные библиотечные функции языка C. Эти функции не входят ни в стандартные классы C++, ни в функции интерфейсов прикладного программирования UNIX и POSIX, но их необходимо знать, так как они бывают весьма полезны при разработке системных приложений.

ГЛАВА 4

Стандартные библиотечные функции С

В языке С определен набор библиотечных функций, которые не имеют прямого соответствия в стандартных классах С++ и интерфейсах прикладного программирования UNIX и POSIX. Эти функции используются для:

- манипулирования данными, их преобразования и шифрования;
- определения пользователями функций с переменным числом аргументов;
- динамического управления памятью;
- представления показаний системных часов в стандартных форматах даты и времени;
- получения системной информации.

Главными преимуществами стандартных библиотечных функций С являются мобильность и низкие затраты на сопровождение пользовательских приложений. Это обусловлено тем, что в большинстве систем (UNIX и другие), которые поддерживают С, применяется один и тот же набор стандартных библиотечных функций С. Эти функции должны иметь одни и те же прототипы и вести себя в разных системах одинаково. Кроме того, библиотечные функции не подвержены частым изменениям, поэтому программы, в которых они используются, легки в сопровождении. Наконец, некоторые из этих библиотечных функций соответствуют стандарту С Американского национального института стандартов (ANSI С), благодаря чему они приемлемы для всех систем, соответствующих этому стандарту. Следовательно, чтобы сократить затраты и время на разработку приложений, рекомендуется использовать библиотечные функции С всякий раз, когда это оказывается возможным.

В этой главе описываются основные библиотечные функции, определенные в стандарте ANSI C, и некоторые библиотечные функции, не относящиеся к этому стандарту, но широко доступные во всех UNIX-системах. Мы подробно описываем эти функции для того, чтобы пользователи с их помощью могли сократить время разработки своих приложений, улучшить переносимость программ и упростить их сопровождение.

Если переносимость программ и простота их сопровождения являются решающими факторами для разрабатываемого вами приложения, то рекомендуется максимально использовать стандартные классы C++ и стандартные библиотечные функции C, а системные интерфейсы прикладного программирования применять только в случае необходимости. Если же ваши приложения должны отличаться высоким быстродействием или если они требуют интенсивного взаимодействия с ядром, то целесообразней будет использовать системные API.

Стандартные библиотечные функции C объявляются в наборе файлов заголовков, которые в UNIX-системах обычно расположены в каталоге */usr/include*. Архивы и совместно используемые библиотеки, которые содержат объектный код данных библиотечных функций,— это соответственно *libc.a* и *libc.co*. В UNIX-системах указанные библиотеки обычно находятся в каталоге */usr/lib*.

В нескольких следующих разделах описываются библиотечные функции ANSI C, определенные в файлах заголовков, перечисленных ниже:

- <stdio.h>
- <stdlib.h>
- <string.h>
- <memory.h>
- <malloc.h>
- <time.h>
- <assert.h>
- <stdarg.h>
- <getopt.h>
- <setjmp.h>

Кроме указанных, в большинстве UNIX-систем есть файлы заголовков, которые не определены в ANSI C:

- <pwd.h>
- <grp.h>
- <crypt.h>

В этих файлах заголовков объявляются функции, которые помогают получить доступ к информации о бюджетах пользователей и групп в UNIX-системах. В указанных системах они определены в библиотеке *libc.a*. Мы описываем эти заголовки в расчете на то, что пользователи сочтут их полезными для разработки приложений.

4.1. <stdio.h>

В файле заголовков <stdio.h> объявляется тип данных FILE, который используется в C-программах для обозначения потоковых файлов, или просто потоков, т.е. файлов, обмен с которыми осуществляется с помощью функций потокового ввода-вывода. Имеется также набор макрокоманд и функций, предназначенных для манипулирования потоковыми файлами. Ниже приведены некоторые из этих макрокоманд и функций, которые уже должны быть знакомы читателям.

Потоковая функция или макрокоманда	Назначение
fopen	Открывает поток для чтения и (или) записи
fclose	Закрывает поток
fread	Читает блок данных из потока
fgets	Читает строку текста из потока
fscanf	Читает форматированные данные из потока
fwrite	Записывает блок данных в поток
fputs	Записывает строку текста в поток
fprintf	Записывает форматированные данные в поток
fseek	Перемещает указатель чтения или записи в потоке
ftell	Возвращает текущую позицию в потоке, начиная с которой будет выполнена следующая операция чтения или записи. Возвращаемое значение — это количество байтов смещения относительно начала потока
freopen	Повторно использует указатель потока для ссылки на новый файл
fdopen	Открывает потоковый файл с указанным дескриптором
feof	Макрокоманда, которая возвращает ненулевое значение, если в данном потоке обнаружен символ конца файла, в противном случае — нулевое значение
ferror	Макрокоманда, которая возвращает ненулевое значение, если в данном потоке была обнаружена ошибка или символ конца файла, в противном случае — нулевое значение
clearerr	Макрокоманда, которая сбрасывает флаг наличия ошибок в данном потоке
fileno	Макрокоманда, которая возвращает дескриптор данного потокового файла

Функция *freopen* часто используется для переназначения стандартного ввода или стандартного вывода выполняемой программы. Прототип этой функции выглядит следующим образом:

```
FILE* freopen ( const char* file_name, const char* mode, FILE* old_stream );
```

Аргумент *file_name* является путевым именем нового потока, который необходимо открыть. Аргумент *mode* определяет, будет ли новый поток открыт для чтения и (или) записи. Это тот же аргумент, который используется в *fopen*. Новый поток должен быть открыт в режиме, не противоречащем режиму доступа к потоку, на который ссылается аргумент *old_stream*. Например, если старый поток открыт только для чтения, то новый поток необходимо открыть также только для чтения. То же самое касается и ситуации, когда старый поток открыт только для записи или для чтения-записи. Эта функция пытается открыть новый поток с заданным режимом доступа. Если новый поток успешно открыт, старый закрывается и указатель потока *old_stream* теперь обозначает новый поток. Если новый поток не может быть открыт, то *old_stream* все равно закрывается. При успешном выполнении эта функция возвращает значение *old_stream*, а в случае неудачи — значение NULL.

В следующем примере эмулируется команда *cp* (копировать файл) ОС UNIX. Программа принимает в качестве аргументов два путевых имени и копирует содержимое файла, указанного первым аргументом (*argv[1]*), в файл, указанный вторым аргументом (*argv[2]*). Заметьте, что вместо использования двух указателей потоков для обозначения этих двух файлов указатели стандартных потоков *stdin* и *stdout* осуществляют ссылки на исходный и создаваемый файл соответственно через функцию *freopen*. Данные из исходного файла читаются посредством использования библиотечной функции *gets* и записываются в создаваемый файл с помощью библиотечной функции *puts*:

```
#include <stdio.h>
int main( int argc, char *argv[])
{
    if ( argc !=3 ) {
        cerr << "usage:" << argv[0] << " <src> <dest>\n";
        return 1;
    }
    (void)freopen( argv[1],"r",stdin );        /* stdin обозначает
                                                  исходный файл */
    (void)freopen( argv[2],"w",stdout );       /* stdout обозначает
                                                  результирующий файл */
    for ( char buf[256]; gets( buf ); )
      puts( buf );
    return 0;
}
```


Функция *fdopen* преобразует дескриптор файла в указатель потока. Дескрипторы файлов используются в интерфейсах прикладного программирования UNIX для получения доступа к файлам. В отличие от указателей потоков они не обеспечивают буферизации данных. Если пользователи хотят выполнить буферизацию данных ввода-вывода, они могут с помощью этой функции преобразовать дескриптор файла в указатель потока. Прототип функции *fdopen* выглядит следующим образом:

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

Аргумент *file_desc* является тем дескриптором файла, который будет преобразован. Аргумент *mode* задает режим доступа к открываемому потоку. Возможные значения *mode* такие же, как и в случае вызова *fopen*, и они должны быть совместимыми с режимом, в котором был открыт файл с дескриптором *file_desc*. В частности, если файл открыт только для чтения, то значение *mode* должно быть "*r*". Точно так же, если данный файл открыт только для записи, то значением *mode* должно быть "*w*". В случае успешного выполнения команды эта функция возвращает новый указатель потока, а в случае неудачи — указатель NULL. Одной из причин неудачи может быть несовместимость значения аргумента *mode* с режимом доступа к файлу с дескриптором *file_desc*.

Следующий пример функции иллюстрирует один из возможных вариантов реализации функции *fopen* (путем использования *fdopen*):

```
FILE* fopen ( const char* file_name, const char* mode )
{
    int fd, access_mode;
    /* преобразовать mode в целое значение access_mode */
    if (( fd = open(file_name, access_mode, 0666 )) < 0 )
        return NULL;
    return fdopen ( fd, mode );
}
```

В этом примере аргумент *mode* необходимо преобразовать из символьной строки в целочисленный флаг *access_mode*. Далее вызывается API *open*, который открывает файл, заданный аргументом *file_name*, и возвращаемый дескриптор файла сохраняется в *fd*. Функция преобразует *fd* в указатель потока через вызов *fdopen* и возвращает этот указатель потока.

Вызов *fdopen* используется также в других ситуациях, например при реализации функции *popen*. Этот случай будет описан в главе 8.

Наконец, в файле заголовков <stdio.h> также объявляются функции *popen* и *pclose*. Данные функции применяются для выполнения в пользовательской программе команд shell. Это очень удобный вариант выполнения системных функций в пользовательских программах, причем некоторые из этих функций с помощью стандартных библиотечных функций и системных API не реализуются.

Прототипы функций *popen* и *pclose* выглядят следующим образом:

```
FILE* popen ( const char shell_cmd, const char* mode );
int pclose ( FILE* stream_ptr );
```

Аргумент *shell_cmd* функции *popen* — это команда shell, задаваемая пользователем. В качестве аргумента разрешается применять любую команду, которую интерпретатор команд shell может выполнить из командной строки. Пользователи имеют право использовать в команде переназначение ввода, переназначение вывода и конвейеры команд. В ОС UNIX функция *popen* для выполнения команды вызывает интерпретатор Bourne-shell. Аргумент *mode* может иметь значение "*r*" или "*w*". Первое из этих значений говорит о том, что функция возвратит указатель потока для чтения данных со стандартного ввода выполняемой команды, а второе — что она возвратит указатель потока для записи данных на стандартный вывод этой команды. Функция возвращает NULL, если команда не может быть выполнена, а в случае успешного ее выполнения — указатель потока. Отметим, что функция *popen* создает *неименованный канал* для передачи данных между вызывающим ее процессом и выполняемой командой. Неименованные каналы рассматриваются в главе 7.

Функция *pclose* вызывается для закрытия указателя потока, который получен с помощью *popen*. Она также гарантирует надлежащее завершение выполняемой команды. Реализация функций *popen* и *pclose* рассматривается в главе 8, где описываются API процессов ОС UNIX.

Приведенная ниже программа *ps.C* выводит на экран все выполняющиеся в UNIX-системе процессы, принадлежащие пользователю root:

```
/* ps.C */
#include <stdio.h>
int main ()
{
    /* выполнить команду */
    FILE * cmdp = popen( "ps -ef | grep root","r" );
    if ( !cmdp )
    {
        perror ( "popen" );
        return 1;
    }
    char result [256] ;
    /* теперь прочитать выходную информацию команды grep */
    while ( fgets( result, sizeof(result), cmdp ) )
        fputs( result, stdout ); /* отобразить каждую прочитанную
                                    строку */

    pclose( cmdp );          // закрыть поток
    return 0;
}
```


4.2. <stdlib.h>

В заголовке <stdlib.h> объявляется набор функций, служащих для преобразования данных, генерации случайных чисел, получения и установки переменных среды shell, управления выполнением программ и выполнения команд shell. Обычно эти функции объявляются в заголовке <stdio.h>, но так как они не включают в себя манипулирование потоками, стандарт ANSI C группирует их в отдельный заголовок.

Функция *system*, объявленная в заголовке <stdlib.h>, выполняет операцию подобно функции *popen*, за исключением того, что пользователи могут получить доступ к стандартному вводу или выводу выполняемой команды. Прототип функции *system* выглядит следующим образом:

```
int system ( const char* shell_cmd );
```

Аргумент *shell_cmd* является строкой символов, которая содержит задаваемую пользователем команду shell. Команда должна быть из числа тех, которые интерпретатор shell может выполнить из командной строки. В *shell_cmd* разрешается применять переназначение ввода-вывода и конвейеры команд. В ОС UNIX эта функция для выполнения команды вызывает Bourne-shell. В случае успешного выполнения команда возвращает нулевое значение, а в случае неудачного выполнения — ненулевое значение. Так, команды shell (*cd /bin; ls -l | sort -b | wc > /tmp/wc.out*) можно выполнить с помощью следующего оператора:

```
if(system("cd /bin; ls -l | sort -b | wc > /tmp/wc.out")==-1)
        perror("system");
```

Этот оператор выполняет команды так же, как при вводе их с консоли UNIX. Отметим, что поскольку для выполнения команды *shell_cmd* функция *system* вызывает новый *shell*, то произведенные при этом изменение рабочего каталога и установка переменных *shell* после возврата из системного вызова *system* оказываются недействительными.

Приведенная ниже программа *mini-shell.C* эмулирует shell UNIX. Она принимает от пользователя одну или более строк команд. И для каждой вводимой строки вызывает функцию *system*, которая выполняет команду. Программа завершается, когда в стандартном вводе встречается признак конца файла:

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

int main()
{
    char  cmd[256];
    do {
```

```
        /* показать подсказку shell */
        cout << "*> " << flush;
        /* получить пользовательскую команду,*/
        /* завершить прием по символу EOF */
        if (!cin.getline(cmd,256)) break;
        /* выполнить пользовательскую команду */
        if (system(cmd) == -1)
            perror(cmd);
    } while (1);
    /* выйти из mini-shell */
    return 0;
}
```

Перечисленные далее функции определяются в заголовке <stdlib.h> и преобразуют данные из символьных строк в данные других форматов, например double, long, int и т.д.:

```
int    atoi  ( const char* str_val );
double atof  ( const char str_val );
long   atol  ( const char* str_val );
double strtod ( const char* str_val, char** endptr );
long   strtol ( const char* str_val, char** endptr, int radix );
unsigned long strtoul ( const char* str_val, char** endptr, int radix );
```

Каждая из перечисленных функций преобразует строку, указанную в *str_val*, в число определенного формата (float, double, long или unsigned long) и возвращает его значение. Если задан аргумент *endptr* и его значение является адресом указателя символьного типа, то этот указатель устанавливается на ту позицию в строке *str_val*, где завершается ее просмотр. Если преобразование не удается, указатель устанавливается на начало строки *str_val* и функция возвращает нулевое значение. Аргумент *radix* задает основание системы счисления числовой строки, находящейся в переменной *str_val*.

В C и C++ есть соответственно функция *sscanf* и класс *istrstream*, которые выполняют операции, аналогичные производимым перечисленными функциями преобразования. Например, функция *atol* может быть записана одним из следующих методов:

```
/* метод C */
#include <stdio.h>
long atol ( const char* str_val)
{
    long x;
    if (sscanf( str_val, "%ld", &x ) == 1 )
        return x;
    else return 0;
}
/* метод C++ */
#include <strstream.h>
```



```
long atol (const char* str_val )
{
    long x;
    istrstream( str_val, strlen(str_val)+1 ) >> x;
    return x;
}
```

Функции *rand* и *srand*, объявленные в заголовке <stdlib.h>, выполняют генерацию случайных чисел. Их прототипы выглядят следующим образом:

```
int rand ( void );
void srand ( unsigned int seed );
```

Функция *srand* получает от пользователя число *seed* и устанавливает начальную точку для новой последовательности псевдослучайных чисел, которые будут возвращаться при каждом следующем вызове *rand*. Последовательность псевдослучайных чисел, возвращенная функцией *rand*, может быть повторена, если вызвать функцию *srand* с тем же значением *seed*. Если *rand* вызывается перед *srand*, то значение *seed* по умолчанию принимается равным 1. Целые числа, возвращаемые функцией *rand*, находятся в диапазоне от 0 до 2^{15} - 1. Если пользователь желает ограничить возвращаемые псевдослучайные числа диапазоном от 1 до N (где N — любое произвольное положительное целое), то вызов *rand* может быть модифицирован следующим образом:

```
int random_num = rand() % N + 1;
```

Следующая функция возвращает случайное число, которое является уникальным для каждого вызова:

```
#include <time.h>
int get_rand()
{
    srand( (unsigned)time( 0 ) );
    return rand();
}
```

В последнем примере функция *time* объявляется в заголовке <time.h>. Она возвращает целое число, которое обозначает количество секунд, прошедших с 1 января 1970 года до текущего времени. (Более подробно эта функция описана в одном из следующих разделов, где рассматривается заголовок <time.h>.) Так как возвращаемое значение функции *time* уникально для вызова (предполагается наличие как минимум односекундного интервала между двумя последовательными вызовами), то аргумент *seed* функции *srand* также уникален и, следовательно, уникальным является случайное число, возвращаемое функцией *rand*. Случайные числа широко применяются в программах статистической обработки и анализа.

В дополнение к упомянутым функциям, в <stdlib.h> объявляются также функции, которые применяются при завершении выполняемых программ:

```
void  exit    ( int status_code );
int   atexit  ( void (*cleanup_fnptr)(void) );
void  abort   ( void );
```

Функция *exit* должна быть знакома читателям, так как она используется для завершения пользовательских программ (процессов) и возвращает целочисленный код завершения в вызывающий shell. Согласно правилам, принятым в ОС UNIX, нулевое значение аргумента *status_code* означает, что программа выполнена успешно. В противном случае значение *status_code* отлично от нуля.

Функция *atexit* может быть вызвана для выполнения функции, определяемой пользователем. Функция *atexit* используется без аргументов и не возвращает никакого значения. Она вызывается функцией *exit* и предназначена для выполнения завершающих действий содержащего вызовы этих функций процесса. С помощью нескольких вызовов *atexit* в процессе можно зарегистрировать несколько функций, и эти функции вызываются (в порядке, обратном порядку регистрации), когда содержащий их процесс вызывает *exit*.

Функция *abort* вызывается, когда процесс находится в аварийном состоянии. Эта функция завершает процесс и в ОС UNIX обеспечивает создание файла *core*, содержащего образ процесса. Такие файлы весьма полезны для отладки прерванных процессов.

Наконец, в заголовке <stdlib.h> объявляются функция *getenv*, которая позволяет процессу получать значения переменных среды shell, а также функция *putenv*, которая дает возможность установить переменную среды в заданное значение. Однако в ANSI C функция *putenv* не определена, хотя и присутствует в большинстве UNIX-систем. Прототипы функций *getenv* и *putenv* выглядят следующим образом:

```
char*  getenv ( const char* env_name );
int    putenv ( const char* env_def );
```

Значение аргумента *env_name*, используемого при вызове *getenv*,— это символьная строка, содержащая имя переменной среды shell. Функция возвращает значение NULL, если данная переменная среды не определена.

Значение аргумента *env_def* в функции *putenv* — символьная строка, содержащая имя переменной среды, знак равенства и значение, которое присваивается переменной. В случае успешного выполнения эта функция возвращает нулевое значение, в случае неудачи — ненулевое.

В приведенном далее примере выводится значение переменной среды PATH, а затем значение переменной среды CC устанавливается равным *c++*:

```
char* env = getenv( "PATH" );
cout << "\"PATH\" value is:" << env << '\n';
if ( putenv( "CC=c++" ) ) cerr << "putenv of CC failed\n";
```

4.3. <string.h>

В заголовке <string.h> объявляется набор функций, предназначенных для манипулирования символьными строками. Эти функции хорошо известны программистам, работающим на C и C++, и используются почти во всех C-программах, производящих обработку символьных строк. Вот наиболее распространенные строковые функции:

```
int    strlen    ( const char* str );
int    strcmp    ( const char* str1, const char* str2 );
int    strncmp   ( const char* str1, const char* str2, const int n );
char*  strcat    ( char* dest, const char* src );
char*  strncat   ( char* dest, const char* src, const int n );
char*  strcpy    ( char* dest, const char* src );
char*  strncpy   ( char* dest, const char* src, const int n );
char*  strchr    ( const char* str, const char ch );
char*  strrchr   ( const char* str, const char ch );
char*  strstr    ( const char* str, const char* key );
char*  strpbrk   ( const char* str1, const char* delimit);
```

Как используются данные строковые функции показано ниже:

Функция	Назначение
strlen	Возвращает количество символов аргумента *str*, оканчивающегося NULL-символом. NULL-символ в возвращаемом значении не учитывается
strcmp	Сравнивает аргументы *str1* и *str2*. Возвращает 0, если строки одинаковы; в противном случае возвращается ненулевое значение
strncmp	Сравнивает первые *n* символов строковых аргументов *str1* и *str2*. Возвращает 0, если символы совпадают; в противном случае возвращается ненулевое значение
strcat	Присоединяет строку аргумента *src* к строке аргумента *dest*. К результирующей строке *dest* добавляется NULL-символ. Возвращает адрес строки аргумента *dest*
strncat	Присоединяет первые *n* символов строки аргумента *src* к строке аргумента *dest*. К результирующей строке *dest* добавляется NULL-символ. Возвращает адрес строки аргумента *dest*

Функция	Назначение
strcpy	Копирует содержимое строки аргумента *src*, включая завершающий NULL-символ, в *dest*. Возвращает адрес строки аргумента *dest*
strncpy	Заменяет первые *n* символов строки аргумента *dest* строкой аргумента *src*. Если размер строки аргумента *src* равен или больше *n*, то NULL-символ не копируется. Возвращает адрес строки аргумента *dest*
strchr	Ищет в строке *str* первый экземпляр символа *ch*. Возвращает адрес символа *ch* в строке *str* или NULL, если *ch* не найден
strrchr	Ищет в строке *str* последний экземпляр символа *ch*. Возвращает адрес символа *ch* в строке *str* или NULL, если *ch* не найден
strstr	Ищет в строке *str* первый экземпляр символьной строки *key*. Возвращает адрес строки *key* в строке *str* или NULL, если *key* не найдена
strpbrk	Ищет в строке *str* экземпляр любого символа, указанного в аргументе *delimit*. Возвращает адрес совпавшего символа в строке *str* или NULL, если совпадение не обнаружено

В дополнение к указанным рассмотрим еще несколько полезных функций, о которых программисты, работающие на C и C++, как правило, не знают.

4.3.1. strspn, strcspn

Прототипы функций *strspn* и *strcspn* выглядят таким образом:

```
const char* strspn ( char* str, const char* delimit );
const char* strcspn ( char* str, const char* delimit );
```

Функция *strspn* возвращает в *str* число начальных символов, которые совпадают с символами, указанными в аргументе *delimit*. Эта функция полезна для пропуска начальных разделительных символов в строке. В нашем примере возвращается адрес следующего не являющегося пробельным символа во входном аргументе *buf*:

```
#include <string.h>
char* skip_spaces ( char* buf )
{
      return buf + strspn( buf, " \t\n" );
}
```

Функция *strcspn* возвращает число начальных символов в строке *str*, которые не указаны в аргументе *delimit*. Эта функция полезна для поиска следующего разделительного символа в символьной строке. В представленном ниже примере возвращается адрес следующей лексемы во входном аргументе *buf*, отделенной одним из пробельных символов:


```
#include <string.h>
char* get_token ( char* buf )
{
     char* ptr = buf + strspn( buf," \n\t");     /* найти начало
                                                    лексемы */
     char* endptr = ptr + strcspn(ptr," \n\t"); /* найти разделитель
                                                    после лексемы */
     if ( endptr > ptr ) *endptr = '\0';
     if ( *ptr )
          return ptr;                            /* возвратить лексему */
     else return NULL;                           /* конец строки.
                                                    Лексемы нет */
```

4.3.2. strtok

Прототип функции *strtok* выглядит следующим образом:

```
const char* strtok ( char* str, const char* delimit );
```

Эта функция разбивает аргумент *str* на одну или несколько лексем. Лексемы отделяются друг от друга символами, указанными в аргументе *delimit*. Если аргумент *str* является адресом символьной строки, то функция *strtok* возвращает указатель на первую лексему в строке. Если же аргумент *str* имеет значение NULL, то функция возвращает указатель на следующую лексему в ранее заданной строке.

Функция возвращает NULL, если лексем в строке больше нет.

Покажем, как строка может разбиваться на лексемы, разделенные пробельными символами. Каждая полученная лексема направляется на стандартный вывод:

```
#include <iostream.h>
#include <string.h>
int main( int argc, char* argv[] )
{
  while ( --argc > 0 )
   for (char* tok; tok=strtok(argv[argc], " \n\t"); argv[argc]=0;)
        cout << "tok:" << tok <<endl;
}
```

Отметим, что функция *strtok* модифицирует входной аргумент *str* путем замещения в нем разделительных символов после лексем NULL-символом. Пользователи, которые желают повторно использовать символьные строки, подлежащие синтаксическому анализу функцией *strtok*, должны заранее сделать их дополнительные копии.

Ниже показан один из возможных способов реализации функции *strtok*. Отметим, что функция имеет статический указатель *lptr*, обозначающий позицию следующей лексемы в строке. Кроме того, если после лексемы функция находит разделительный символ, то она замещает его NULL-символом.

Таким образом, функция модифицирует входную строку символов, извлекая из нее лексемы.

```
#include <string.h>
char* my_strtok ( char* str, const char* delimit )
{
     static char *lptr;
     if ( str ) (                          /* начать разбор новой
                                              строки */
          str += strspn( str,delimit )     /* пропустить начальные
                                              разделители */
          if (!*str ) return NULL;         /* выход, если это
                                              NULL-строка */
          lptr = str;
     )
     else if ( !lptr )                     /* продолжить разбор старой
                                              строки */
           return NULL;                    /* выход, если лексем больше
                                              нет */

     char* tokn = lptr + strspn( lptr," \t\n");  /* пропустить
                                                    начальный
                                                    разделитель */
     lptr = tokn + strcspn( tokn," \t\n");       /* найти следующий
                                                    разделитель */
     if ( *tokn && lptr > tokn )
          *lptr++= '\0';                  // завершить лексему NULL-символом
     else lptr = NULL;                    /* найти последнюю лексему
                                             в строке */
     return *tokn ? tokn : NULL;          /* возвратить лексему, если
                                             таковая имеется */
}
```

4.3.3. strerror

Прототип функции *strerror* выглядит следующим образом:

```
const char* strerror ( int errno );
```

Эту функцию можно использовать для получения системного диагностического сообщения. Значение аргумента *errno* может быть любым кодом ошибки, определенным в файле заголовков <sys/errno.h>, или глобальной переменной *errno*. Глобальная переменная *errno* устанавливается при каждом вызове системного API. Если API выполнен успешно, ее значение равно нулю, в противном случае оно отлично от нуля.

Возвращаемая строка символов служит только для чтения и не должна переназначаться пользователями.

Функция *perror* может быть вызвана для печати диагностического сообщения, если какой-то системный API завершается неудачей. Функция *strerror* позволяет пользователям определять их собственные версии функции *perror*.

В следующем примере описывается возможная реализация функции *perror* с использованием *strerror*:

```
include <iostream.h>
#include <string.h>

void my_perror ( const char* msg_header )
{
     if (msg_header && strlen(msg_header))     /* печатать, если
                                                  определено
                                                  пользователем */
          cerr << msg_header << ":" << strerror( errno ) << endl;
     else cerr << strerror ( errno ) << endl;
)
/* тест-программа для функции my_perror */
int main( int argc, char* argv[] )
(
     FILE *fp;
     while ( --argc > 0)                  /* для каждого аргумента командной
                                             строки */
          if ( (fp = fopen(*++argv, "r")) )  /* fp=0 при неудачном
                                                открытии файла */
             my_perror( *argv );          /* печатать диагностическое
                                             сообщение */
          else fclose ( fp );             /* закрыть файл, если он открыт
                                             нормально */
     return 0;
}
```

4.4. <memory.h>

В заголовке <memory.h> объявляется набор функций, предназначенных для манипулирования байтовым потоком. Эти функции похожи на строковые, но в отличие от них имеют более широкое назначение и могут использоваться для манипулирования несимвольными строковыми объектами. В частности, данные функции можно применять для инициализации, сравнения и копирования объектов типа *struct*.

Вот функции, объявленные в заголовке <memory.h>:

```
void*  memset  ( const void* memp, int ch, size_t len );
int    memcmp  ( const void* mem1, const void* mem2, size_t len );
void*  memcpy  ( void* dest, const void* src, size_t len );
void*  memccpy ( void* dest, const void* src, const int ch, size_t len );
void*  memchr  ( const void* memp, const int ch, size_t len );
```

Функция *memset* заполняет символом *ch* первые *len* байтов области памяти, на которую указывает *memp*. Она возвращает адрес *memp*.

Следующий пример иллюстрирует инициализацию переменной типа *struct stat* NULL-символами:

```
struct stat *statp = new stat;
if (statp)
      (void)memset( (void*)statp, NULL, sizeof( struct stat ) );
```

Функция *bzero* в BSD UNIX инициализирует всю область памяти NULL-символами. Эта функция может быть реализована через функцию *memset* следующим образом:

```
void bzero ( char *memp, int len )
{
      (void)memset( memp, NULL,(size_t)len );
}
```

Функция *memcmp* сравнивает первые *len* байтов из двух областей памяти, на которые указывают аргументы *mem1* и *mem2*. Эта функция возвращает ноль, если первые *len* байтов этих областей памяти идентичны, положительное значение — если область *mem1* содержит символы, которые лексикографически больше символов в *mem2*, отрицательное значение — если область *mem1* содержит символы, которые лексикографически меньше символов в *mem2*.

Приведенные ниже операторы проверяют равенство двух переменных типа *struct stat*:

```
int cmpstat ( struct stat* statpl, struct stat* statp2 )
{
  return memcmp((void*)statpl, (void*)statp2, sizeof(struct stat));
}
```

Функция *strcmp* может быть реализована с помощью функции *memcmp* следующим образом:

```
int my_strcmp (const char* strl, const char* str2 )
{
    int lenl = strlen( strl ), len2 = strlen( str2 );
    if ( lenl > len2 ) lenl = len2;
    return memcmp( (void*)strl, (void*)str2, lenl );
}
```

Более того, в BSD UNIX функция *bcmp* тоже может быть реализована через функцию *memcmp*:

```
#define bcmp ( sl, s2, n )  memcmp( (void*)sl, (void*)s2, (size_t)n)
```

Функция *memcpy* копирует первые *len* байтов данных из области памяти, на которую указывает *src*, в область памяти, на которую указывает *dest*. Эта функция возвращает адрес области памяти *dest*.

Функция *bcopy* в операционной системе BSD UNIX может быть также реализована с помощью функции *memcpy*:

```
#define bcopy(src, dest, n) memcpy((void*)dest, (void*)src, (size_t)n)
```

Заметим, что функции *bcopy*, *bcmp* и *bzero* не определены в стандарте ANSI C, но они широко используются в UNIX-программах.

Функция *memcpy* может быть использована для реализации функции *strcpy*:

```
#define strcpy( dest, src )        \
     memccpy( (void*)dest, (void*)src, '\0', (size_t)strlen( src ) )
```

Функция *memccpy* копирует данные из области памяти, на которую указывает аргумент *src*, в область памяти, на которую указывает аргумент *dest*. Функция либо копирует данные до первого появления символа, указанного в аргументе *ch* (включая и этот символ), либо копирует первые *len* байтов.

Функция *memccpy* может быть использована для реализации функции *strncpy*:

```
#define strncpy( dest, src, n )    \
     memccpy( (void*)dest, (void*)src, '\0', (size_t)n )
```

Функция *memchr* производит поиск в первых *len* байтах области памяти, на которую указывает аргумент *memp*, и возвращает адрес первого экземпляра символа *ch* в этой области или NULL, если *ch* не найден. Функция *memchr* может быть использована для реализации функции *strchr*.

```
#define strchr( str,ch ) memchr( (void*)str, ch, (size_t)strlen( str ))
```

4.5. <malloc.h>

В заголовке <malloc.h> объявляется набор функций, предназначенных для динамического распределения и освобождения памяти. Программисты, работающие на C++, редко используют эти функции, применяя вместо них операторы *new* и *delete*, выполняющие те же действия. Тем не менее функция *realloc*, объявленная в заголовке <malloc.h>, может быть использована для динамического регулирования объема памяти, а оператор *new* в C++ такой возможности не предоставляет. Использование функции *realloc* подробно описывается ниже.

В заголовке файла <malloc.h> объявляются такие функции:

```
void*  malloc  (const size_t size );
void*  calloc  (const size_t num_record, const size_t size_per_record );
void   free    (void* memp );
void*  realloc (void* old_memp, const size_t new_size );
```

Пользователи уже должны быть знакомы с тем, как применяются функции *malloc*, *calloc* и *free*. В частности, оба следующих оператора выделяют динамическую память объемом 1048 байтов:

```
char* mem1 = (char*)malloc( 1048 ) );          // стиль C
char* nem2 = new char [ 1048 ];                // стиль C++
```

Функция *calloc* подобна функции *malloc*, за исключением того, что она гарантирует инициализацию каждого элемента выделенной памяти значением 0.

Функция *free*, как и оператор *delete*, используется для освобождения динамической памяти. Представленный ниже оператор освобождает динамическую память, на которую указывает переменная *mem1*:

```
free (mem1);
```

Функция *realloc* используется для регулирования размера динамической памяти, выделенной функцией *malloc* или *calloc*. Это очень полезно при управлении массивом, который может изменять свой размер в ходе выполнения процесса. Например, программа, сохраняющая данные, которые вводятся пользователем со стандартного ввода, не может знать наперед, сколько строк данных будет получено от пользователя. В такой ситуации с помощью функции *realloc* можно обеспечить динамическое выделение именно того объема памяти, который нужен для хранения вводимых пользователем данных в каждый текущий момент времени.

Заметим, что в рассматриваемом случае для сохранения пользовательских входных данных можно применять связный список или массив фиксированного размера. По сравнению с массивом (статическим или размещаемым динамически) связный список имеет некоторые недостатки. Поскольку каждая запись связного списка требует памяти для следующего указателя, то он занимает больший объем памяти, чем массив, при том же количестве элементов. Более того, создание и обработка связных списков, как правило, требует больше времени, чем выборка данных из массивов. Недостатком массива фиксированного размера в сравнении с массивом, размещаемым динамически, является то, что он требует предварительного выделения всей памяти для сохранения максимально возможного количества входных данных — следовательно, он менее эффективен в использовании памяти, чем динамически размещаемые массивы. Кроме того, при использовании массива фиксированного размера устанавливается ограничение на максимально возможное количество входных данных, что может оказаться весьма нежелательным для пользователей. При использовании динамически размещаемого массива подобные ограничения не возникают. То есть динамически размещаемый массив оказывается более предпочтительным, чем связный список, в тех случаях, когда доступ к хранимым данным производится часто и порядок их не меняется. Кроме того, динамически размещаемый массив предпочтительнее статического еще и потому, что его размер может изменяться (уменьшаться или увеличиваться) со временем, а задание верхнего предела размера оказывается непрактичным.

Функция *realloc* принимает два аргумента. Первый аргумент, *old_memp*, содержит адрес предварительно выделенной области динамической памяти, второй, *new_size* — размер новой области динамической памяти в байтах. Значение *new_size* может быть больше или меньше размера старой области памяти, обозначенной аргументом *old_memp*. Функция *realloc* пытается отрегулировать размер старой области динамической памяти под *new_size*. Если

это не получается, то выделяется новая область динамической памяти *new_size*. Затем функция копирует максимально возможный объем данных из старой области памяти в новую, и старая область памяти освобождается. Если размер новой области памяти больше размера старой, то содержимое памяти в новой области, не инициализированное данными старой памяти, не определяется.

В приведенной ниже программе *malloc.C* показано использование функции *realloc* для сохранения пользовательских входных данных, поступающих со стандартного ввода:

```
#include <iostream.h>
#include <string.h>
#include <malloc.h>
#include <malloc.h>
int main()
{
     char** inList = 0, buf[256];
     int numIn = 0, max_size = 0;
     /* получить от пользователя все входные строки */
     while (cin.getline(buf, sizeof(buf))){
        if ( ++numIn > max_size ){
              max_size += 2;
              if ( !inList )
                 inList = (char**)calloc(max_size,sizeof(char*) );
              else
                 inList = (char**)realloc(inList,sizeof(char*)*max_size );
           }
           /* сохранить входную строку */
           inList[numIn-1] = (char*)malloc( strlen( buf )+1 );
           strcpy( inList[numIn-1], buf );
     }
     /* напечатать все входные строки пользователя */
     while ( --numIn >= 0 ){
           cout << numIn << ": " << inList[numIn] << endl;
           free( inList[numIn] );
     }
     free ( inList );
     return 0;
}
```

Эта программа читает строки со стандартного ввода. По мере того как программа считывает каждую строку, строка сохраняется в массиве *inList*. Размер массива *inList* регулируется динамически на основании фактического числа прочитанных входных строк. Переменные *max_size* и *numIn* содержат соответственно текущий размер массива *inList* и число фактически прочитанных входных строк. Если значение *numIn* совпадает с *max_size* и читается новая входная строка, то значение *max_size* увеличивается на два, а размер массива *inList* увеличивается на два элемента путем вызова функции *realloc*.

После того как будут прочитаны все входные строки и в стандартном входном потоке встретится признак конца файла, программа пошлет все

сохраненные строки на стандартный вывод в порядке, обратном порядку чтения, и по пути освободит всю динамическую память.

Отметим, что в этом примере первое выделение памяти массиву *inList* производится через вызов *calloc*. Это объясняется тем, что если попытаться выделить динамическую память с помощью функции *realloc*, т.е. таким образом:

```
char* memp = (char*)realloc( 0, new_size )
```

то в некоторых UNIX-системах будет выдано сообщение об ошибке, связанной с сегментацией (например, в Sun OS 4.1.x), тогда как в других системах функция *realloc* будет работать. Следовательно, для обеспечения переносимости программ необходимо избегать присваивания значения NULL первому аргументу при любом вызове *realloc*.

Наконец, динамическая память, выделенная вызовами *malloc*, *calloc* и *realloc* должна освобождаться только функцией *free*, а память, выделенная оператором *new*,— только оператором *delete*.

4.6. <time.h>

В заголовке <time.h> объявляется набор функций, предназначенных для вызова системных параметров времени. Они могут применяться для определения местного времени и даты, времени и даты в универсальном формате (UTC), а также статистических данных об использовании процессами времени центрального процессора.

В заголовке <time.h> объявляются функции:

```
time_t       time      (time_t* timv);
const char*  ctime     (const time_t* timv);
struct tm*   localtime (const time_t* timv);
struct tm*   gmtime    (const time_t* timv);
const char*  asctime   (const tm* tm_p);
time_t       mktime    (struct tm* tm_p);
clock_t      clock     (void);
```

Функция *time* возвращает количество секунд, прошедших со дня официального рождения ОС UNIX — 1 января 1970 года. Результат типа *time_t* хранится в возвращаемом значении функции и по адресу *timv*, если он не равен NULL.

Функция *ctime* возвращает значения местного времени и даты в следующем формате:

```
"Sun Sept. 16 01:03:52 1997\n"
```


Функция *ctime* почти всегда используется с функцией *time* для получения значений местного времени и даты:

```
time_t timv = time ( 0 );
cout << "local time:" << ctime( &timv );
```

Функции *localtime* и *gmtime* получают адрес переменной типа *time_t* и возвращают адрес записи типа *struct tm*, предназначенной только для чтения. Принимаемый аргумент типа *time_t* должен быть установлен предыдущим вызовом функции *time*, и возвращаемая запись типа *struct tm* содержит информацию о времени и дате соответственно по местному времени и в формате UTC. Запись типа *struct tm* может быть передана в функцию *asctime* для получения символьной строки в том же формате, что и у возвращаемого значения *ctime*. Тип данных *struct tm* определен в заголовке <time.h>.

Приведенные ниже операторы иллюстрируют использование этих функций:

```
#include <time.h>
time_t timv = time( 0 );
struct tm *local_tm = localtime( &timv );
struct tm *gm_tm = gmtime( &timv );
cout << "local time stamp:" << asctime(local_tm );
cout << "UTC time stamp:" << asctime( gm_tm );
```

Следующая функция возвращает значение, соответствующее моменту времени, отстоящему от текущего на указанное количество часов:

```
const char* time_stamp(long offset_hours )
{
     time_t timv = time( 0 );          // получить текущее время
     timv += offset_hours * 60 * 60;   // преобразовать смещение
                                       // в секунды
     return ctime( &timv );            // вернуть новое значение времени
}
```

Функция *mktime* противоположна функциям *localtime* и *gmtime*. Она получает адрес записи типа *struct tm* и возвращает для нее значение типа *time_t*. Кроме того, функция *mktime* нормализует выходные данные, приводя их к общепринятому виду для любой произвольной даты в формате UTC (начиная с 00:00:00 1 января 1970 года по 03:14:07 UTC 19 января 2038 года включительно).

Покажем программу, с помощью которой можно вычислить, на какой день недели приходится 5 апреля 1999 года:

```
#include <iostream.h>
#include <time.h>
static struct tm time_str;          // инициализировать все поля в 0
main()
{
     time_t tmv;
     time_str.tm_year = 1999 - 1900; // год = 1999
     time_str.tm_mon = 4 - 1;        // месяц = April
     time_str.tm_mday = 5;           // день = 5
```

```
    if ( tmv=mktime(&time_str) ) != -1 ) {
        timv[3] = NULL;
        cout << timv << endl;      // должна напечатать "Mon"
    }
}
```

Возвращаемое значение функции *clock* может варьироваться в зависимости от системы. Стандарт ANSI C определяет, что функция *clock* возвращает число микросекунд, прошедших с момента начала выполнения вызывающего ее процесса. Однако в некоторых системах UNIX возвращаемым значением функции *clock* является количество микросекунд, прошедших с момента первого вызова процессом функции *clock*. Точное определение возвращаемого результата пользователи могут найти в руководстве программиста или на man-странице, посвященной функции *clock*, в своей системе.

На примере программы *clock.C* покажем, как правильно использовать функцию *clock* для контроля времени выполнения процесса независимо от того, как функция реализована в данной системе:

```
#include <iostream.h>
#include <time.h>

main()
{
    time_t clock_tick = CLOCKS_PER_SEC;
    clock_t start_time = clock();          // включить таймер

    /* выполнить требуемые вычисления...*/
    clock_t elapsed_time = clock() - start_time;
    cout << "Run time: " << (elapsed_time / clock_tick) << endl;
    return 0;
}
```

4.7. <assert.h>

В заголовке <assert.h> объявляется макрокоманда, используемая для проверки некоторых условий выполнения процесса, которые в нормальной ситуации всегда должны быть истинны. Если все же во время выполнения процесса условие не выполняется, то макрокоманда выводит сообщение об ошибке в стандартный поток ошибок с указанием той строки исходного файла, в которой нарушается проверяемое условие. После этого макрокоманда прерывает процесс.

Таким образом, макрокоманда *assert* помогает сэкономить время, уходящее на отладку программы, проверяя условия типа "не должно было произойти". Эта макрокоманда может быть изъята из готового продукта путем указания при компиляции программы ключа -DNDEBUG.

В приведенной ниже программе *assert.C* демонстрируется использование макрокоманды *assert*:


```
#include <fstream.h>
#include <string.h>
#include <assert.h>
int main( int argc, char* argv[] )
{
    assert ( argc > 1 );          // должен быть хотя бы 1 аргумент
    ifstream ifs( argv[1] );
    assert( ifs.good() );         // поток должен открываться нормально
    char *nam = new char[strlen(argv[1])+1];
    assert( nam );                // не должно быть значения NULL
    return 0;
}
```

Когда программа *assert.C* компилируется и выполняется без аргумента, на консоль выводится сообщение:

```
% CC assert.C -o assert; assert
Assertion failed: file "assert.C", line 5
```

Макрокоманда *assert* определена в <assert.h> следующим образом:

```
#ifndef NDEBUG
#define assert(ex) ( if (!(ex)) (\
fprintf(stderr,"Assertion failed: file: \"%s\", line %d\n",\
        _FILE_, _LINE_); exit(1);}
#endif
```

Заметим, что в данном случае *assert* является макрокомандой и может компилироваться отдельно от программы пользователя путем определения макроса NDEBUG.

4.8. <stdarg.h>

В заголовке <stdarg.h> объявляется набор макрокоманд, которые могут быть использованы для определения функций с переменным количеством аргументов. Примерами таких функций в C являются *printf* и *scanf*. Данные функции могут быть вызваны с одним или несколькими аргументами, и для нормальной работы вызываемые функции должны обработать все аргументы. Это достигается путем использования макрокоманд, указанных в заголовке <stdarg.h>.

В заголовке <stdarg.h> определяются следующие макрокоманды:

```
#define va_start(ap,parm) (ap) = (char*)(&(parm) + 1)
#define va_arg(ap,type) ((type*)((char*)(ap) += sizeof(type)))[-1]
#define va_end(ap)
```

Чтобы использовать перечисленные макрокоманды, функция должна иметь в своем прототипе один четко определенный аргумент. Макрокоманда *va_start* вызывается той или иной функцией для установки аргумента *ap* так, чтобы он указывал на позицию в динамическом стеке, где находится следующее значение аргумента после *parm* (который является последним из известных аргументов вызывающей функции). Макрокоманда делает это путем добавления байтового смещения к адресу данных, на которые указывает *parm*. Так определяется адрес следующего после *parm* значения аргумента функции.

Макрокоманда *va_arg* вызывается, когда необходимо извлечь следующее значение аргумента в динамическом стеке. Для того чтобы эта команда сработала, вызывающий процесс должен знать тип данных следующего аргумента в стеке. Макрокоманда *va_arg* выполняет для каждого вызова два действия:

- изменяет *ap* так, чтобы он указывал на позицию в стеке после следующего аргумента, подлежащего возврату в вызывающую функцию;
- возвращает следующий аргумент в стеке.

Для первой установки *ap* выполняется операция

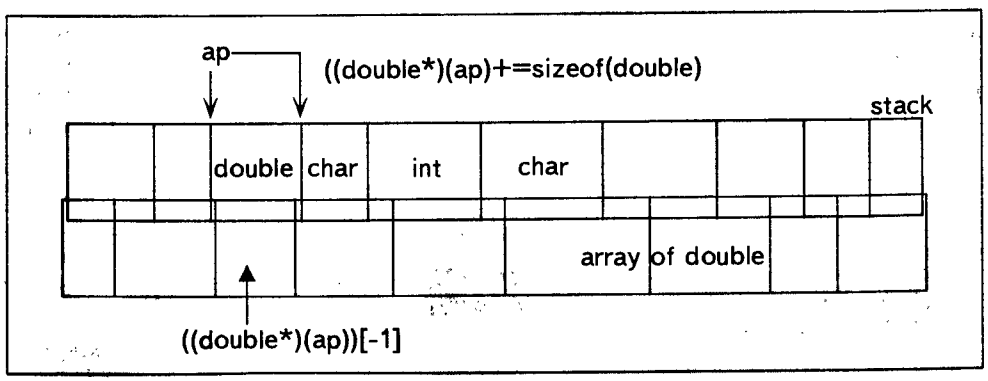
```
(char*)(ap) += sizeof(type)
```

которая изменяет тип указателя *ap* на символьный, а затем увеличивает его значение на размер следующего аргумента. Это, по сути дела, смещает *ap* так, чтобы он указывал на адрес аргумента, следующего за тем, который подлежит выборке.

Второе задание выполняется с помощью операции:

```
*(type*)( (char*)(ap) += sizeof(type)) )[-1]
```

Данная операция вновь изменяет *ap* так, чтобы он указывал на массив типа *type*. Индекс *-1* обеспечивает возвращение вызывающей функции следующего аргумента в стеке. Принцип действия макрокоманды *va_arg* показан на приведенной ниже схеме; предполагается, что аргумент, который должен быть возвращен, имеет тип *double*.



Макрокоманда *va_end* в настоящее время является макроопределением пустой операции. Она задана как парная для макрокоманды *va_start* и предназначена для перспективного расширения функциональных возможностей <stdarg.h>.

Программа *printf.C* содержит функцию *my_printf*, которая эмулирует функцию *printf*:

```
#include <iostream.h>
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include <floatingpoint.h>

#define SHOW(s) fputs(s,stdout), cnt+=strlen(s)

/* моя версия printf */
int my_printf ( const char* format, ...)
{
     char *tokp, ifmt[256], *hdr = ifmt, buf[256];
     int cnt = 0;
     va_list pa;
     strcpy(ifmt,format);   // получить копию введенного текста
     va_start(pa,format);   // pa — указатель на аргументы в стеке

     while (tokp=strchr(hdr,'%')) {  /* поиск символа "%" */
          *tokp++ = '\0';
          SHOW(hdr);                        // показывает текст до "%"
          if (strchr("dfgeisc%",*tokp)) {   /* выполнять, если символ
                                               формата является
                                               допустимым */
            switch (*tokp) {
             case 'd':      // %i, %d
             case 'i':
               gconvert((double)va_arg(pa,int),sizeof(buf),0,buf);
               break;
             case 's':      // %s
               strcpy(buf,va_arg(pa,char*));
               break;
             case 'c':      // %c
                buf[0] = va_arg(pa,char);
                buf[1] = '\0';
                break;
             case 'f':     // %f
                gconvert(va_arg(pa,double),8,1,buf);
                break;
             case 'g':     // %g
                gconvert(va_arg(pa,double),8,0,buf);
                break;
             case '%':     // %%
                strcpy(buf,"%");
                break;
               }
               SHOW(buf);        // показать извлеченный аргумент
          }
```

```
        else {                          /* формат задан неверно. Показать
                                           символ как он есть */
            putchar(*tokp);
            cnt++;
        }
        hdr = tokp+1;
    }
    SHOW(hdr);      // показать, если он есть, завершающий текст
    va_end(pa);
    return cnt;     // возвратить число напечатанных символов
}

int main()
{
    int cnt = my_printf("Hello %% %s %zZZ\n", "world");
    cout << "No. char: " << cnt << endl;
    cnt = printf("Hello %% %s %zZZ\n", "world");
    cout << "No. char.: " << cnt << endl << endl;
    cnt = my_printf("There are %d days in %c year\n", 365, 'A');
    cout << "No. char.: " << cnt << endl;
    cnt = printf("There are %d days in %c year\n", 365, 'A');
    cout << "No. char.: " << cnt << endl << endl;
    cnt = my_printf("%g x %i = %f\n", 8.8, 8, 8.8*8);
    cout << "No. char.: " << cnt << endl;
    cnt = printf("%g x %i = %f\n", 8.8, 8, 8.8*8);
    cout << "No. char.: " << cnt << endl << endl;
    return 0;
}
```

В приведенной программе функция *gconvert* преобразует значение типа double в формат символьной строки. Эта функция не входит в стандарт ANSI C, но обычно доступна в большинстве систем UNIX. Прототип функции *gconvert* выглядит следующим образом:

```
char* gconvert ( double dval, int ndigits, int trailing, char* buf );
```

Аргумент *dval* в функции *gconvert* содержит подлежащее преобразованию значение типа double, а аргумент *buf* указывает на определяемый пользователем буфер, в который помещается преобразованная символьная строка. Аргумент *ndigits* задает максимальное число значащих цифр, которые может содержать буфер *buf*, а значением аргумента *trailing* может быть 0 или 1, что определяет, будет или нет отброшена последняя десятичная точка или 0. Эта функция возвращает адрес буфера, указанный в аргументе *buf*.

Компиляция и пробный запуск этой тест-программы дали такие результаты:

```
% CC printf.C -o printf
% printf
Hello % world zZZ
No. of char: 18
```



```
Hello % world zZZ
No. of char: 18

There are 365 days in A year
No. of char: 29
There are 365 days in A year
No. of char: 29

8.8 x 8 = 70.400000
No. of char: 20
8.8 x 8 = 70.400000
No. of char: 20
```

С макросом *va_arg* связаны функции *vfprintf*, *vsprintf*, *vprintf*. Они подобны соответственно функциям *fprintf*, *sprintf* и *printf*, за исключением того, что принимают *ap* как указатель на аргументы вызывающих функций. Прототипы этих функций представлены ниже:

```
int vprintf  ( const char* format, va_list ap );
int vsprintf ( char* buf, const char* format, va_list ap );
int vfprintf ( FILE* fp, const char* format, va_list ap );
```

Данные функции могут быть использованы для создания универсальной функции выдачи сообщений:

```
/* source file: test_vfprintf.C */
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
typedef enum { INFO, WARN, ERROR, FATAL } MSG_TYPE_ENUM;
static int numErr, numWarn, numInfo;

void msg ( const MSG_TYPE_ENUM mtype, void* format, ... )
{
    switch (mtype)
        case INFO:      numInfo++;
                        break;
        case WARN:      numWarn++;
                        fputs( "Warning: ",stderr );
                        break;
        case ERROR:     numErr++;
                        fputs( "Error: ",stderr );
                        break;
        case FATAL:     fputs( "Fatal: ", stderr );
                        break;
    }
    va_list ap;
    va_start( ap, format );
    vfprintf( stderr, (char*)format, ap );
    va_end( ap );
```

```
    if (mtype == FATAL ) exit( 2 );
)

/* тест-программа для функции msg */
int main()
{
    msg( INFO, "Hello %% %s %%\n", "world" );
    msg( WARN, "There are %d days in %c year\n", 365, 'A' );
    msg( ERROR, "%g x %i = %f\n", 8.8, 8, 8.8*8 );
    msg( FATAL, "Bye-bye\n" );
    return 0;
)
```

Компиляция и выполнение контрольной программы дали следующие результаты:

```
% CC test_vfprintf.C -o test_vfprintf
% test_vfprintf
Hello % world %
Warning: There are 365 days in A year
Error: 8.8 x 8 = 70.400000
Fatal: Bye-bye
```

4.9. Аргументы командной строки и ключи

Функция *getopt*, которая объявляется в заголовке <stdlib.h>, может быть использована для реализации программ, которые принимают UNIX-подобные ключи и аргументы командной строки. Синтаксис вызова таких программ должен быть следующим:

```
<имя_программы> [-<ключ> ...] [<аргумент> ...]
```

Все ключи (или опции) должны начинаться с символа "-" и состоять из одной буквы, например: *-o*. При этом учитывается и регистр. Несколько ключей можно объединять (так, ключи *-a*, *-b* можно дать как *-ab* или *-ba*). За ключом может следовать только с ним связанный необязательный аргумент (например, *-o a.out*). Если два или несколько ключей объединены, то только последний из них может принимать такой аргумент. Например, *-o a.out -O* может быть записано как *-Oo a.out*, но не как *-oO a.out*.

Если аргумент не связан с ключом, то после него ключи ставить нельзя. Таким образом, следующий вызов не верен, поскольку перед ключом *-o* указан не связанный с ним аргумент */usr/prog/test.c*.

```
% a.out  -l /usr/prog/test.c -o abc
```

Если при вызове программы указанные правила соблюдаются, то с помощью функции *getopt* ключи и все связанные с ними аргументы могут быть получены из командной строки. Эта возможность будет показана ниже.

Функция *getopt* и связанные с ней глобальные переменные *opterr*, *optarg* и *optind* объявляются в заголовке <stdlib.h>:

```
extern int      optind, opterr;
extern char*    optarg;

int getopt ( int argc, char* const* argv[], const char* optstr;
```

Первыми двумя аргументами функции *getopt* являются переменные *argc* и *argv* функции *main*. Аргумент *optstr* содержит список букв ключей, которые допустимы для данной программы. Функция просматривает вектор *argv* и ищет ключи, которые определены в *optstr*. После каждого вызова *getopt* функция возвращает букву ключа, указанную в *optstr* и найденную в *argv*. Если ключ определен в *optstr* как *<switch_letter>:*, то этот ключ, если он найден, должен сопровождаться аргументом, который может быть получен через глобальный указатель *optarg*.

Если ключ находится в *argv*, но не указан в *optstr*, то функция *getopt* направит сообщение в стандартный поток ошибок и возвратит символ "?". Если же пользователь перед вызовом *getopt* устанавливает глобальную переменную *opterr* в ненулевое значение, то последующие недопустимые ключи, найденные в *argv*, функция будет игнорировать.

Наконец, если в *argv* больше нет ключей, то функция *getopt* возвращает значение EOF и переменная *optind* устанавливается так, чтобы указывать на элемент *argv*, где хранится первый неключевой аргумент командной строки. Если *optind* совпадает с *argc*, то в программе не связанных с ключом аргументов нет.

Следующая программа *test_getopt.C* принимает ключи *-a*, *-b* и *-o*. Если указан ключ *-o*, то с ним должно быть задано имя файла:

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
static char* outfile;
static int a_flag, b_flag;

int main( int argc, char* argv[] )
{
    int ch;
    while ( (ch=getopt( argc, argv,"o:ab" )) != EOF )
        switch ( ch )  {
            case 'a':       a_flag = 1;     // найден -a
                break;
            case 'b':       b_flag = 1;     // найден -b
                break;
            case 'o':       outfile = new char[strlen(optarg)+1];
                strcpy(outfile,optarg);  // найдено -o <имя файла>
```

```
                break;
            case '?':    /* getopt выдаст сообщение об ошибке */
            default:    break;    // недопустимый ключ
    }
    /* ключей больше нет. Просмотреть остальные неключевые
       аргументы */
    for ( ; optind < argc; optind++ )
         cout << " non-switch argument: " << argv[optind] << endl;

    return 0;
}
```

Компиляция и пробные прогоны программ *test_getopt.C* дали следующие результаты:

```
% CC test_getopt.C -o test_getopt
% test_getopt
% test_getopt -abo xyz /etc/hosts
non_switch argument: /etc/hosts
% test_getopt -xay -bz /usr/lib/libc.a
test_getopt: illegal option — x
test_getopt: illegal option — y
test_getopt: illegal option — z
non_switch argument: /usr/lib/libc.a
```

Функция *getopt* сопровождается следующими ограничениями: ключи должны состоять только из одной буквы; ключи должны иметь либо связанные с ними аргументы, либо не иметь их вообще; пользователи не имеют права определять ключи, которые могут иногда принимать, а иногда не принимать аргументы; функция не проверяет тип данных ключей; пользователи не могут указывать взаимоисключающие ключи.

Несмотря на указанные недостатки, *getopt* позволяет значительно сэкономить время, уходящее на разработку и отладку программ, а также обеспечивает соответствие пользовательских программ правилам вызова, принятым в ОС UNIX.

4.10. <setjmp.h>

В заголовке <setjmp.h> объявляется набор функций, которые позволяют процессу вызывать оператор перехода *goto* из одной функции в другую. Вызов С-оператора *goto* позволяет процессу передать управление выполнением от одного оператора к другому лишь в рамках этой же функции. Функции, определенные в заголовке <setjmp.h>, устраняют данное ограничение. Эти функции необходимо использовать лишь тогда, когда без них действительно нельзя обойтись. Например, если ошибка обнаружена в рекурсивной функции, то есть смысл сообщить об ошибке, а затем выполнить оператор перехода (*goto*) в основную функцию, то есть как бы начать процесс сначала. Именно это делает shell UNIX при обнаружении ошибки в одном из его процессов. При таких обстоятельствах функции <setjmp.h> обеспечивают

эффективное восстановление после сбоев и освобождают пользователя от необходимости добавлять все новые и новые уровни программ контроля ошибок для их исправления. Тем не менее, как и при использовании *goto*, возникает одна проблема: если эти функции используются в программе нерационально, то пользователю бывает трудно отследить ход выполнения программы.

В заголовке <setjmp.h> определяются следующие функции:

```
int  setjmp  ( jmp_buf loc );
void longjmp ( jmp_buf loc, int val );
```

Функция *setjmp* регистрирует позицию в коде программы, куда будет возвращать управление будущий оператор *goto* (посредством вызова *longjmp*). Тип данных *jmp_buf* определяется в заголовке <setjmp.h>, а в аргументе *loc* регистрируется позиция оператора *setjmp*. Если пользователь желает определить в программе несколько точек, в которые может вернуться будущий вызов *longjmp*, то каждая точка должна быть зарегистрирована в переменной типа *jmp_buf* и ее позиция должна быть определена с помощью функции *setjmp*.

Функция *setjmp* всегда возвращает 0, если она вызывается непосредственно в процессе для сохранения состояния стека и регистрации точки возврата.

Функция *longjmp* вызывается для передачи управления в позицию, указанную в аргументе *loc*. Код программы, отмеченный этой точкой, должен находиться в функции, имеющейся среди функций, которые вызывают текущую функцию. Когда процесс "перепрыгивает" в эту целевую функцию, все содержимое стека, используемое текущей и вызывающими ее функциями, функцией *longjmp* отбрасывается. Процесс возобновляет выполнение, повторно выполняя оператор *setjmp* в целевой функции, которая отмечена аргументом *loc*. Возвращаемым значением функции *setjmp* в этом случае является *val*, указанное в вызове функции *longjmp*. Значение *val* не должно быть нулевым (если это 0, то функция *setjmp* устанавливает его в 1), чтобы с его помощью можно было указать, где и почему была вызвана функция *longjmp*, и выполнить необходимую обработку ошибок.

На примере программы *test_setjmp.C* покажем, как могут использоваться функции *setjmp* и *longjmp*:

```
/* source file name: test_setjmp.C */
#include <iostream.h>
#include <setjmp.h>
static jmp_buf loc;
int main()

{
     int retcode, foo();
     if ( (retcode=setjmp( loc )) != 0 ) { // устранение ошибок
          cerr << "Get here from longjmp. retcode=" << retcode << endl;
```

```
        return 1;
    }
    /* нормальный ход выполнения программы */
    cout << "Program continue after setting loc via setjmp...\n";
    foo();
    cout << "Should never get here ....\n";
    return 1;
}
int foo()
{
    cout << "Enter foo. Now call longjmp....\n";
    longjmp (loc, 5);
    cout << "Should never gets here.....\n";
    return 2;
}
```

Компиляция и пробное выполнение программы *test_setjmp.C* дают следующие результаты:

```
% CC test_setjump.C -o test_setjmp
% test_setjmp
Program continue after setting loc via setjmp...
Enter foo. Now call longjmp...
Get here from longjmp. return code=5
```

4.11. <pwd.h>

В заголовке <pwd.h> определяется набор функций, предназначенных для получения учетной информации о пользователях, содержащейся в UNIX-файле */etc/passwd*. Здесь объявляются следующие функции:

```
const struct passwd* getpwnam (const char* user_name );
const struct passwd* getpwuid (const int uid );
int setpwent ( void );
const struct passwd* getpwent (void );
int endpwent ( void );
const struct passwd* fgetpwent (FILE* fptr );
```

Тип данных *struct passwd* определен в <pwd.h> так:

```
struct passwd
{
    char*     pw_name;        // регистрационное имя пользователя
    char*     pw_passwd;      // зашифрованный пароль
    int       pw_uid;         // идентификатор пользователя
    int       pw_gid;         // идентификатор группы
    char*     pw_age;         // информация о минимальном сроке
                              // действия пароля
    char*     pw_comment;     // общая информация о пользователе
    char*     pw_dir;         // начальный каталог пользователя
```



```
    char*     pw_shell;       // регистрационный shell пользователя
};
```

В каждой записи *struct passwd* содержится одна строка данных файла */etc/password*. В этой строке хранится информация о бюджете пользователя в UNIX-системе. В частности, это регистрационное имя пользователя, назначенный идентификатор пользователя, идентификатор группы, регистрационный shell, начальный каталог, регистрационный пароль (в зашифрованном виде) и т.д.

Функция *getpwnam* принимает регистрационное имя пользователя в качестве аргумента и возвращает указатель на запись типа *struct passwd*, которая содержит информацию об этом пользователе, если он имеет право работать в системе. Если заданное имя пользователя недействительно, функция возвращает NULL-указатель.

Вывести на экран начальный каталог пользователя с именем *joe* можно следующим образом:

```
struct passwd *pwd = getpwnam( "joe" );
if ( !pwd )
    cerr << "'joe' is not a valid user on this system\n";
else
    cout << pwd->pw_name << ",home=" << pwd->pw_dir << endl;
```

Функция *getpwuid* принимает в качестве аргумента идентификатор пользователя и возвращает указатель на запись типа *struct passwd*, которая содержит информацию об этом пользователе, если для него в системе создан бюджет. Если заданный идентификатор пользователя недействителен, функция возвращает NULL-указатель.

Определить имя и регистрационный shell пользователя с идентификатором 15 можно таким образом:

```
struct passwd *pwd = getpwuid( 15 );
if ( !pwd )
    cerr << "'15' is not a valid UID on this system\n";
else
    cout << pwd->pw_name << ",shell=" << pwd->pw_shell << endl;
```

Функция *setpwent* устанавливает указатель чтения на начало файла */etc/passwd*. Функция *getpwent* смещает этот указатель на следующую запись файла */etc/passwd*. Когда с помощью функции *getpwent* будет осуществлен просмотр всех записей файла */etc/passwd*, она возвратит NULL-указатель, что обозначает конец файла. Для закрытия файла */etc/passwd* вызывается функция *endpwent*.

Следующая программа, *test_pwd.C*, посылает на стандартный вывод перечень всех определенных в системе пользователей с идентификаторами пользователей и групп:

```
#include <iostream.h>
#include <pwd.h>
int main()
```

```
{
     setpwent();
     for ( struct passwd *pwd; pwd=getpwent(); )
          cout  << pwd->pw_name << ", UID: " << pwd->pw_uid
                << ", GID: " << pwd->pw_gid << endl;
     endpwent();
     return 0;
}
```

Функция *fgetpwent* подобна функции *getpwent*, с той лишь разницей, что в ней пользователь задает указатель на файл с такой же структурой, как у файла */etc/passwd*. Эта функция возвращает пользовательские учетные данные, находящиеся в следующей записи файла. Функции *setpwent* и *endpwent* с функцией *fgetpwent* не используются.

4.12. <grp.h>

В заголовке <grp.h> определяется набор функций, предназначенных для получения учетной информации о группах, содержащейся в UNIX-файле */etc/group*. Здесь объявляются следующие функции:

```
const struct group* getgrnam (const char* group_name );
const struct group* getgrgid (const int gid );
int setgrent ( void );
const struct group* getgrent (void );
int endgrent ( void );
const struct group* fgetgrent (FILE* fptr );
```

Тип данных *struct group* определен в файле заголовков <grp.h> так:

```
struct group
{
     char*     gr_name;        // имя группы
     char*     gr_passwd;      // зашифрованный пароль группы
     int       gr_gid;         // идентификатор группы
     char**    gr_comment;     // имена членов группы
};
```

Каждая запись *struct group* содержит данные из одной строки файла */etc/group*. Эта строка содержит информацию о бюджете одной группы в UNIX-системе. В частности, это имя группы, назначенный идентификатор группы и список имен входящих в нее пользователей.

Функция *getgrnam* принимает в качестве аргумента имя группы и возвращает указатель на запись типа *struct group*, которая содержит информацию о группе, если эта группа определена в системе. Если заданное имя группы недействительно, функция возвращает NULL-указатель.

Определить идентификатор группы *developer* можно следующим образом:

```
struct group *grp = getgrnam( "developer" );
if ( !grp )
     cerr << "'developer' is not valid group on this system\n";
else
     cout << grp->gr_name <<", GID=" << grp->gr_gid << endl;
```

Функция *getgrgid* принимает в качестве аргумента идентификатор группы и возвращает указатель на запись типа *struct group*, которая содержит информацию об этой группе, если группа определена в системе. Если заданный идентификатор группы недействителен, функция возвращает NULL-указатель.

Информацию о членах группы с идентификатором 200 можно получить с помощью следующего выражения:

```
struct group *grp = getgrgid( 200 );
if ( !grp )
     cerr << "'200' is not valid GID on this system\n";
else for ( int i=0; grp->pw_comment && grp->pw_comment[i]; i++ )
     cout << grp->gr_comment[i] << endl;
```

Функция *setgrent* устанавливает указатель чтения файла на начало файла */etc/group*. Функция *getgrwent* смещает этот указаталь на следующую запись файла */etc/group*. Когда с помощью функции *getgrwent* будет осуществлен просмотр всех записей в каталоге */etc/passwd*, она возвратит NULL-указатель, что обозначает конец файла. Для закрытия файла */etc/group* вызывается функция *endgrent*.

Программа *test_grp.C* направляет на стандартный вывод перечень всех зарегистрированных в системе групп с их идентификаторами:

```
#include <iostream.h>
#include <grp.h>
int main()
{
     setgrent();
     for ( struct group *grp; grp=getgrent(); )
          cout << grp->gr_name << ", GID: " << grp->gr_gid <<endl;
     endgrent();
     return 0;
}
```

Функция *fgetgrent* подобна функции *getgrent*, за исключением того что в ней пользователь задает указатель на файл, имеющий такой же синтаксис, что и файл */etc/group*. Эта функция возвращает учетные данные группы, находящиеся в текущей записи данного файла. Функции *setgrent* и *endgrent* с функцией *fgetgrent* не используются.

4.13. <crypt.h>

В заголовке <crypt.h> объявляется набор функций, предназначенных для шифрования и дешифрования данных. Это очень важные функции, обеспечивающие безопасность системы. Например, файлы пользовательских паролей и системных данных, которым необходима высокая степень защиты, должны быть зашифрованы так, чтобы ни один человек, не имеющий специального разрешения, не мог узнать, что они из себя представляют. Более того, чтобы читать и изменять эти объекты, уполномоченные лица должны знать секретные ключи дешифровки.

В заголовке <crypt.h> объявляются следующие функции:

```
char* crypt    (const char* key, const char* salt );
void  setkey   (const char salt[64] );
void  encrypt  (char key[64], const int flag );
```

Функция *crypt* используется в UNIX-системах для шифрования пользовательских паролей и проверки действительности регистрационного пароля пользователя. Эта функция принимает два аргумента. Первый аргумент, *key*, является паролем, определенным пользователем. Второй аргумент, *salt*, применяется для идентификации полученной зашифрованной строки. Значение аргумента *salt* состоит из двух символов, взятых из следующего набора:

```
a-z, A-Z, 0-9, /
```

Если функция *crypt* вызывается процессом *password* для шифрования нового пароля пользователя, то процесс возвращает случайное значение *salt*. Полученная строка, зашифрованная и снабженная уникальным идентификатором, имеет следующий формат:

```
<salt><зашифрованная строка пароля>
```

Затем, когда пользователь пытается зарегистрироваться в системе, указывая имя и пароль, регистрационный процесс проверяет аутентичность пользователя следующим образом:

```
#include <iostream.h>
#include <crypt.h>
#include <pwd.h>
#include <string.h>
int check_login( const char* user_name, const_char* password)
{
     struct passwd* pw;
     if ( !(pw=getpwnam( user_name )) ) {
          cerr << "Invalid login name:" << user_name << endl;
          return 0;       // не аутентичен
     }
     char* new_pw = crypt( password, pw->pw_passwd );
```



```
     if ( strcmp( new_pw,pw->pw_passwd ) ) {
          cerr << "Invalid password:" << password << endl;
          return 0;      //не аутентичен
     }
/* имя пользователя и пароль действительны */
     return 1
}
```

В этом примере функция *check_login* вызывается для того, чтобы убедиться, что указанное регистрационное имя пользователя и пароль действительны. Функция возвращает 1, если они действительны, 0 — если нет. Данная функция вызывает функцию *getpwnam* для преобразования указанного имени пользователя в указатель на запись типа *struct passwd*. Если имя пользователя действительно, то функция *getpwnam* возвращает значение, отличное от NULL. Если нет — она возвращает значение NULL, а функция *check_login* возвращает код ошибки.

После получения записи типа *struct passwd* функция *check_login* для шифрования данного пароля вызывает функцию *crypt*. Аргумент *salt*, указанный в вызове *crypt*, находится в поле *pw_passwd* записи типа *struct passdw*. Первыми двумя символами зашифрованного пароля пользователя являются символы, заданные аргументом *salt*, который был использован для создания зашифрованного пароля. Возвращаемое значение *crypt* является зашифрованной строкой пароля, и оно сравнивается со значением *pw_passwd*. Если они совпадают, то данный пароль действителен, в противном случае функция *check_login* возвращает код ошибки.

Как следует из сказанного, функция *crypt* не расшифровывает строки, тем не менее она может быть использована для шифрования новой строки и последующего ее сравнения со старой с целью проверки содержания последней. Необходимо отметить, что первые два символа шифрованной строки — это символы, заданные аргументом *salt*, который использовался для ее создания. Если две одинаковые строки зашифровать с различными значениями *salt*, то зашифрованные строки будут различаться.

Функции *setkey* и *encrypt* выполняют действия, аналогичные действию функции *crypt*, однако они используют алгоритм шифрования данных по стандарту DES Национального бюро стандартов США (NBS), который более надежен, чем алгоритм, используемый функцией *crypt*. Аргумент функции *setkey* — это массив символов, состоящий из 64 элементов. Каждый из этих элементов должен содержать целое значение 1 или 0, которое является одним битом из восьмибайтового значения *salt*. Первый аргумент *key* функции *encrypt* является символьным массивом, состоящим из 64 элементов. Каждый из этих элементов содержит один бит из восьмибайтового значения *key*, которое должно быть зашифровано (если значение третьего аргумента *flag* — 0) или расшифровано (если значение *flag* — 1). Полученная зашифрованная или расшифрованная строка передается обратно в вызывающую функцию, изменяя значение ее аргумента *key*. Функция *encrypt* может обрабатывать максимум восемь символов за один вызов.

4.14. Заключение

В этой главе описываются библиотечные функции ANSI C и некоторые библиотечные функции C, характерные для ОС UNIX. Стандартными классами C++ и интерфейсами прикладного программирования UNIX и POSIX они не охватываются. Зная эти функции, пользователь может сократить время разработки приложений, добиться высокого уровня переносимости программного обеспечения и значительно уменьшить затраты на сопровождение конечного продукта. Возможность использования некоторых из этих библиотечных функций продемонстрирована на ряде примеров.

Как уже было сказано, описанные библиотечные функции имеют один недостаток: их ассортимент и возможности недостаточны для разработки приложений системного уровня. Для создания таких приложений необходимо использовать интерфейсы прикладного программирования UNIX и POSIX. Эти API описываются в других главах нашей книги, где освещается методика их использования в системном программировании и приводятся примеры применения для реализации некоторых стандартных библиотечных функций C, упомянутых в настоящей главе.

ГЛАВА 5

Интерфейсы прикладного программирования UNIX и POSIX

В UNIX-системах имеется набор функций API (широко известных как *системные вызовы*), которые могут вызываться программами пользователей для выполнения специфических системных задач. Эти функции позволяют пользовательским приложениям непосредственно манипулировать системными объектами типа файлов и процессов, чего нельзя сделать, используя только стандартные библиотечные функции C. Кроме того, многие команды UNIX, библиотечные функции C и стандартные классы C++ (например, класс *iostream*) вызывают эти интерфейсы прикладного программирования для фактического выполнения необходимой работы. Таким образом, с помощью этих API пользователи могут избежать издержек, связанных с вызовом библиотечных функций C и стандартных классов C++, и создавать свои собственные версии команд UNIX, библиотечных функций C и классов C++.

В большинстве UNIX-систем имеется общий набор API для решения следующих задач:

- определения конфигурации системы и получения информации о пользователях;
- манипулирования файлами;
- создания процессов и управления ими;
- осуществления межпроцессного взаимодействия;
- обеспечения связи по сети.

Большинство API ОС UNIX обращаются к внутренним ресурсам ядра UNIX. Так, когда один из этих API вызывается процессом (процесс —

программа пользователя, находящаяся в состоянии выполнения), контекст выполнения процесса переключается ядром из режима пользователя в режим ядра. Режим пользователя — обычный режим выполнения любого пользовательского процесса. Он обеспечивает процессу доступ только к его собственным данным. Режим ядра — защищенный режим выполнения, который позволяет процессу пользователя получать ограниченный доступ к данным ядра. Когда выполнение API завершается, пользовательский процесс переключается обратно в режим пользователя. Это переключение контекста для каждого вызова API гарантирует, что доступ процессов к данным ядра контролируется, и уменьшает вероятность того, что вышедшее из-под контроля пользовательское приложение сможет повредить всю систему. В общем, вызов API требует больше времени, чем вызов пользовательской функции, из-за необходимости переключения контекста. Таким образом, для приложений, требующих высокого быстродействия, пользователи должны вызывать системные API только в том случае, если это абсолютно необходимо.

5.1. Интерфейсы прикладного программирования POSIX

Большинство API, определенных в стандартах POSIX.1 и POSIX.1b, соответствуют API ОС UNIX. Тем не менее комитеты POSIX все же разрабатывают собственные API, когда ощущается нехватка API UNIX. Например, комитет POSIX.1b создает новый набор API для межпроцессного взаимодействия на основе сообщений, разделяемой памяти и семафоров. Аналогичные механизмы для межпроцессного взаимодействия есть и в ОС UNIX System V, но последние не используют путевые имена для определения этих средств межпроцессного взаимодействия, и процессы не могут использовать их для осуществления связи по локальной сети. Поэтому комитет POSIX.1b создал другую версию сообщений, разделяемой памяти и семафоров, в которой упомянутые недостатки устранены.

И все же API POSIX используются и ведут себя аналогично API UNIX. Однако пользователи должны определять _POSIX_SOURCE (для API POSIX.1) и (или) _POSIX_C_SOURCE (для API POSIX.1 и POSIX.1b) в своих программах, чтобы обеспечить объявление API POSIX в файлах заголовков, указанных в этих программах.

5.2. Среда разработки UNIX и POSIX

В файле заголовков <unistd.h> объявляются некоторые широко используемые API POSIX.1 и UNIX. Имеется также набор относящихся к API заголовков, размещенных в каталоге <sys> (в UNIX-системе это каталог */usr/include/sys*). Заголовки типа <sys/...> объявляют специальные типы данных для объектов данных, которыми манипулируют и API, и пользовательские процессы. Кроме того, заголовок <stdio.h> объявляет функцию *perror*, которая может быть вызвана процессом пользователя всякий раз, когда

выполнение API завершается неудачей. Функция *perror* обеспечивает вывод на экран определенного системой диагностического сообщения для любого отказа, вызванного API.

Большая часть объектного кода API POSIX.1, POSIX.1b и UNIX находится в библиотеках *libc.a* и *libc.so*. Таким образом, не нужно определять никаких специальных ключей компилятора, чтобы указать, в каком архиве или совместно используемой библиотеке хранится объектный код API. В некоторых системах, однако, объектный код определенных сетевых коммуникационных API находится в специальных библиотеках (например, в ОС Solaris 2.X фирмы Sun Microsystems API гнезд (sockets) хранятся в библиотеках *libsocket.a* и *libsocket.so*). Поэтому пользователи должны обращаться к руководству системного программиста за информацией о специальных заголовках и библиотеках, необходимых для API, которые они используют.

5.3. Общие характеристики интерфейсов прикладного программирования

Хотя API POSIX и UNIX выполняют от лица пользователей разнообразные системные функции, большинство из них возвращает целое число, значение которого свидетельствует о корректности завершения выполнения. В частности, если API возвращает -1, это значит, что попытка выполнить API потерпела неудачу и глобальной переменной *errno* (которая объявлена в заголовке <errno.h>) присваивается код ошибки. Пользовательский процесс может вызывать функцию *perror*, чтобы направить диагностическое сообщение об отказе на стандартный вывод, или функцию *strerror*, задав ей в качестве аргумента значение *errno*. Функция *strerror* возвращает строку диагностического сообщения, и пользовательский процесс может вывести это сообщение так, как он считает нужным (например, поместить его в файл регистрации ошибок).

Коды ошибок, которые могут быть присвоены *errno* любым API, определены в файле заголовков <errno.h>. Когда пользователь просматривает man-страницу API, на ней, как правило, перечисляются возможные коды ошибок, которые присваиваются *errno* данным API, и причины их возникновения. Так как эта информация всегда доступна пользователям, а коды ошибок в различных системах могут быть разными, в этой книге не будут подробно описываться значения *errno* для отдельных API. Ниже приведен список наиболее распространенных кодов ошибок и их значения.

Код ошибки	Значение
EACCESS	Процесс не имеет права на выполнение операции с помощью API
EPERM	API был прерван, потому что вызывающий процесс не имеет права привилегированного пользователя
ENOENT	Интерфейсу прикладного программирования было указано недействительное имя файла

Код ошибки	Значение
BADF	API был вызван с недействительным дескриптором файла
EINTR	Выполнение API было прервано каким-то сигналом (сигналы и вызываемые ими прерывания рассматриваются в главе 9)
EAGAIN	API был прерван, потому что какой-то необходимый системный ресурс был временно недоступен. Этот API следует вызвать позже
ENOMEM	API был прерван потому, что ему не была выделена динамическая память
EIO	В процессе выполнения API произошла ошибка ввода-вывода
EPIPE	API попытался записать данные в канал, для которого не существует читающего процесса
EFAULT	Интерфейсу прикладного программирования был передан недействительный адрес в одном из аргументов
ENOEXEC	API не смог выполнить программу с помощью одного из системных вызовов семейства *exec*
ECHILD	Процесс не имеет ни одного порожденного процесса, завершения которого он мог бы ждать

Если API выполнен успешно, он возвращает нулевое значение или указатель на некоторую запись данных, где хранится необходимая пользователю информация.

5.4. Заключение

В этой главе дан краткий обзор интерфейсов прикладного программирования UNIX и POSIX, описана методика их использования, приведены общие характеристики. Эти API достаточно мощны и дают возможность пользователям разрабатывать развитые системные программы, манипулирующие системными объектами (например, файлами и процессами) более разнообразными способами, нежели это позволяют делать стандартные библиотечные функции C и классы C++. Кроме того, пользователи могут применять эти API для создания собственной библиотеки или классов C++ и собственных версий shell-команд, чтобы расширить ассортимент команд. Следует отметить, однако, что в большинстве API выполняется переключение контекста пользовательских процессов между режимом пользователя и режимом ядра, поэтому применение этих API приводит к увеличению затрат времени.

В остальных главах книги API UNIX и POSIX исследуются более подробно. Предназначены они для манипулирования файлами и процессами, обеспечения межпроцессного взаимодействия и удаленного вызова процедур. Приводятся примеры использования API для создания пользовательских версий библиотечных функций C и shell-команд, а также для построения классов C++ с целью создания обобщенных типов данных для системных объектов (например, процессов) и системных функций (например, функций, обеспечивающих межпроцессное взаимодействие).

ГЛАВА 6

Файловая система ОС UNIX

Файлы — основа любой операционной системы, поскольку именно с ними производится наибольшее число действий. Когда вы выполняете команду в системе UNIX, ее ядро выбирает соответствующий исполняемый файл из файловой системы, загружает текст инструкций в память и создает процесс для выполнения команды от вашего имени. Кроме того, в ходе выполнения процесс может читать данные из файлов и записывать их в файлы. Создание операционной системы всегда начинается с разработки эффективной системы управления файлами.

Файлы в UNIX- и POSIX-системах могут быть самых разных типов. Здесь используются текстовые файлы, двоичные файлы, файлы каталогов и файлы устройств. Кроме того, в UNIX- и POSIX-системах имеется набор общих системных интерфейсов, предназначенных для осуществления доступа к файлам, благодаря чему прикладные программы могут работать с ними по единой методике. Это, в свою очередь, упрощает задачу разработки прикладных программ для данных систем.

В этой главе будут рассмотрены файлы различных типов, используемые в UNIX- и POSIX-системах, и будет показано, как они создаются и применяются. Кроме того, существует набор общих атрибутов, которые операционная система хранит для каждого файла, имеющегося в системе. Данная глава содержит подробное описание этих атрибутов и варианты их использования. Наконец, описываются структуры данных ядра UNIX System V и всех процессов, которые используются при манипулировании файлами и для формирования интерфейса системных вызовов. Системные вызовы UNIX и POSIX.1, применяемые для управления файлами, описаны в следующей главе.

6.1. Типы файлов

В UNIX- и POSIX-системах существуют файлы следующих типов:

- обычный файл;
- каталог;
- FIFO-файл;
- байт-ориентированный файл устройства;
- блок-ориентированный файл устройства.

Обычный файл может быть текстовым или двоичным. В UNIX- и POSIX-системах эти типы файлов не различаются и оба могут быть "исполняемыми" при условии, что для них установлено разрешение на выполнение и они могут читаться или записываться пользователями, имеющими соответствующие права доступа.

Обычные файлы могут создаваться, просматриваться и модифицироваться с помощью различных средств (например, текстовых редакторов и компиляторов) и удаляться определенными системными командами (например, командой *rm* в ОС UNIX).

Каталог подобен папке, которая содержит много файлов, а также подкаталоги. Каталог предоставляет пользователям средства для организации их файлов в некую иерархическую структуру, основанную на взаимосвязи файлов и направлений их использования. Например, UNIX-каталог */bin* содержит все системные исполняемые программы, такие как *cat*, *rm*, *sort* и т.д.

Каталог может быть создан в UNIX с помощью команды *mkdir*. Следующая UNIX-команда создаст каталог */usr/foo/xyz*, если он не существует:

```
mkdir /usr/foo/xyz
```

Каталог в ОС UNIX считается пустым, если он не содержит никаких других файлов, кроме ссылок на текущий и родительский каталоги, и может быть удален посредством команды *rmdir*. Следующая UNIX-команда удаляет каталог */usr/foo/xyz*, если он существует и если он пустой:

```
rmdir /usr/foo/xyz
```

Содержимое каталога в UNIX может быть отображено на экране с помощью команды *ls*.

Блок-ориентированный файл устройства служит для представления физического устройства, которое передает данные блоками. Примерами блок-ориентированных устройств являются дисководы жестких дисков и дисководы гибких дисков. *Байт-ориентированный файл устройства* служит для представления физического устройства, которое передает данные побайтово. Примерами байт-ориентированных устройств могут служить построчно-печатающие принтеры, модемы, консоли. Физическое устройство могут представлять и блок-ориентированные, и байт-ориентированные файлы, что обеспечивает доступ к ним различными методами. Например, байт-ориентированный файл жесткого диска используется для передачи данных между процессом и диском без обработки (не поблочно).

Прикладная программа может выполнять операции чтения и записи с файлом устройства так же, как с обычным файлом, а операционная система будет автоматически вызывать соответствующий драйвер устройства для выполнения фактической передачи данных между физическим устройством и приложением.

Обратите внимание на то, что на физическое устройство могут ссылаться и байт-ориентированный, и блок-ориентированный файлы. Прикладная программа таким образом выбирает необходимый метод обмена данными с этим устройством (посимвольный, через байт-ориентированный файл устройства, или поблочный, через блок-ориентированный файл).

Файл устройства создается в ОС UNIX с помощью команды *mknod*. Ниже указанная команда создает байт-ориентированный файл устройства с именем */dev/cdsk0*; старшим и младшим номерами устройства являются 115 и 5 соответственно. Аргумент *c* определяет, что файл, который будет создан, является байт-ориентированным файлом устройства:

```
mknod /dev/cdsk   c   115   5
```

Старший номер устройства — это индекс в таблице ядра, которая содержит адреса всех драйверов устройств, известных системе. Всякий раз, когда процесс читает данные из файла устройства или записывает в него, ядро, используя старший номер файла, выбирает и вызывает драйвер устройства для осуществления фактического обмена данными с физическим устройством. Младший номер устройства — это целое значение, которое передается как аргумент в драйвер устройства при его вызове. Младший номер устройства сообщает драйверу устройства, с каким конкретно физическим устройством он взаимодействует (драйвер может обслуживать физические устройства нескольких типов) и какая схема буферизации ввода-вывода должна использоваться при обмене.

Драйверы устройств инсталлируются либо поставщиками физических устройств, либо поставщиками операционных систем. При инсталляции драйвера устройства ядро операционной системы должно быть реконфигурировано. Эта схема позволяет расширять операционную систему при необходимости ввода устройств новых типов, которые становятся нужны пользователю.

Блок-ориентированный файл устройства также создается в ОС UNIX командой *mknod*, однако вторым аргументом команды *mknod* должен быть не *c*, а *b*. Аргумент *b* показывает, что создаваемый файл является блок-ориентированным файлом устройства. С помощью приведенной ниже команды создается блок-ориентированный файл устройства */dev/bdsk* со старшим и младшим номерами устройства — 287 и 101 соответственно:

```
mknod  /dev/bdsk b 287 101
```

В ОС UNIX команда *mknod* должна выполняться привилегированным пользователем. Кроме того, в UNIX все файлы устройств, как правило, помещаются в каталог */dev* или в его подкаталог.

FIFO-файл — это специальный файл, предназначенный для организации канала и создания временного буфера, обеспечивающего взаимодействие двух или более процессов посредством записи данных в этот буфер и чтения данных из буфера. В отличие от обычных файлов размер буфера, связанного с FIFO-файлом, установлен равным PIPE_BUF. (PIPE_BUF и его минимальное значение в POSIX.1, _POSIX_PIPE_BUF, определены в заголовке <limits.h>.) Процесс может записывать в FIFO-файл больше, чем PIPE_BUF байтов данных, но буфер после заполнения может быть заблокирован. В этом случае процесс должен подождать, пока читающий процесс прочитает данные из канала и освободит место, необходимое для завершения операции записи. Наконец, доступ к данным в буфере осуществляется по алгоритму "первым пришел — первым вышел", поэтому такой файл и называется FIFO-файлом (FIFO — first in, first out).

Буфер, связанный с FIFO-файлом, создается, когда первый процесс открывает этот файл для чтения или записи. Буфер удаляется, когда все процессы, которые связаны с FIFO-файлом, закрывают свои ссылки (например, указатели на потоки) на FIFO-файл. Таким образом, данные, записанные в FIFO-буфер, являются временными; они сохраняются до тех пор, пока есть процесс, установивший прямую связь с FIFO-файлом для выборки данных.

FIFO-файл может быть создан в UNIX командой *mkfifo*. Приведенная ниже команда создает FIFO-файл с именем */usr/prog/fifo_pipe*, если он не существует:

```
mkfifo /usr/prog/fifo_pipe
```

В некоторых ранних версиях ОС UNIX (например, в UNIX System V.3), FIFO-файлы создавались с помощью команды *mknod*. Указанная ниже UNIX-команда создает FIFO-файл */usr/prog/fifo_pipe*, если он не существует:

```
mknod /usr/prog/fifo_pipe p
```

Для создания FIFO-файлов в UNIX System V.4 можно использовать и команду *mknod*, и команду *mkfifo*, тогда как BSD UNIX поддерживает только команду *mkfifo*.

FIFO-файл может быть удален подобно любому обычному файлу. Так, FIFO-файл может быть удален с помощью команды *rm*.

Кроме упомянутых типов файлов, в BSD UNIX и UNIX System V.4 определяется также так называемая ***символическая ссылка***. Символическая ссылка содержит путевое имя, которое обозначает другой файл в локальной или удаленной файловой системе. В POSIX.1 символические ссылки еще не поддерживаются, хотя уже предложено добавить этот тип файлов в будущие версии стандарта.

Символическая ссылка может быть создана в UNIX командой *ln*. Следующая команда создает символическую ссылку */usr/mary/slink*, которая ссылается на файл */usr/jose/original*. Команда *cat* обеспечит вывод содержимого файла */usr/jose/original* на экран:


```
ln   -s  /usr/jose/original /usr/mary/slink
cat  -n  /usr/mary/slink
```

Путевое имя, на которое ссылается символическая ссылка, может быть получено в UNIX с помощью команды *ls -l*, выполняемой над этим файлом-ссылкой. Следующая команда покажет, что */usr/mary/slink* является символической ссылкой на файл */usr/jose/original*:

```
% ls  -l /usr/mary/slink
lr--r--r--  1 terry     18 Aug 20, 1994 slink -> /usr/jose/original
%
```

Можно создать символическую ссылку, указывающую на другую символическую ссылку. Если символические ссылки задаются как аргументы в UNIX-командах *vi*, *cat*, *more*, *head*, *tail* и т.д., эти команды будут обрабатывать символические ссылки с целью получить доступ к обычным файлам, которые обозначены такими ссылками. При этом UNIX-команды *rm*, *mv* и *chmod* будут оперировать только аргументами-символическими ссылками, а не файлами, которые они обозначают.

6.2. Файловые системы UNIX и POSIX

Файлы в UNIX- и POSIX-системах хранятся в древовидной иерархической файловой системе. Корень файловой системы — это корневой каталог, обозначенный символом "/". Каждый промежуточный узел в дереве файловой системы — это каталог. Конечные вершины дерева файловой системы являются либо пустыми каталогами, либо не каталогами.

Абсолютное путевое имя файла состоит из имен всех каталогов, ведущих к указанному файлу, начиная с корневого каталога. Имена каталогов в путевом имени отделяются друг от друга символами "/". Так, путевое имя */usr/xyz/a.out* означает, что файл *a.out* расположен в каталоге *xyz*, который, в свою очередь, находится в каталоге *usr*, а каталог *usr* — в каталоге "/".

Относительное путевое имя может состоять из символов "." и "..". Они являются ссылками соответственно на текущий и родительский каталоги. Например, путевое имя *../../.login* обозначает файл с именем *.login*, который находится в каталоге двумя уровнями выше текущего каталога. Хотя POSIX.1 не требует, чтобы каталог содержал файлы "." и "..", в нем все-таки указано, что относительные путевые имена с символами "." и ".." должны интерпретироваться так же, как в ОС UNIX.

Длина имени файла не может превышать NAME_MAX символов, а общее число символов путевого имени не должно превышать PATH_MAX. Установленные в POSIX.1 минимальные значения NAME_MAX и PATH_MAX — это _POSIX_NAME_MAX и _POSIX_PATH_MAX соответственно (определены в заголовке <limits.h>).

Кроме того, в POSIX.1 определен набор символов, который должен поддерживаться всеми POSIX.1-совместимыми операционными системами как допустимый для имен файлов. Прикладные программы, которые

переносятся в системы POSIX.1 и UNIX, должны манипулировать файлами с именами, состоящими из следующих символов:

```
A-Z    a-z    0-9    _    -    .
```

Путевое имя файла называется *жесткой ссылкой*. Файл может обозначаться несколькими путевыми именами, если пользователь создаст одну или несколько жестких ссылок на него с помощью UNIX-команды *ln*. В частности, следующая UNIX-команда создает новую жесткую ссылку */usr/prog/new/n1* для файла */usr/foo/path1*. После выполнения этой команды к файлу можно обращаться по любому из этих путевых имен.

```
ln /usr/foo/path1 /usr/prog/new/n1
```

Обратите внимание, что если в данной команде указана опция *-s*, ссылка */usr/prog/n1* будет символической, а не жесткой. Различия между жесткими и символическими ссылками будут разъяснены в главе 7.

Ниже перечислены файлы и каталоги, которые, как правило, определены в большинстве систем UNIX, хотя стандарт POSIX.1 не считает их обязательными.

Файл	Что содержит
/etc	Системные административные файлы и программы
/etc/passwd	Всю информацию о пользователях
/etc/shadow	Пароли пользователей (только для UNIX System V)
/etc/group	Всю информацию о группах
/bin	Все системные программы, в частности cat, rm, cp и т.д.
/dev	Все байт-ориентированные и блок-ориентированные файлы устройств
/usr/include	Стандартные файлы заголовков
/usr/lib	Стандартные библиотеки
/tmp	Временные файлы, создаваемые программами

6.3. Атрибуты файлов в UNIX и POSIX

И в UNIX, и в POSIX.1 поддерживается набор общих атрибутов для всех файлов, присутствующих в файловой системе. Эти атрибуты описаны ниже.

Атрибут	Значение
file type (тип файла)	Тип файла
access permission (права доступа)	Права доступа к файлу для владельца, группы и прочих пользователей
hard link count (счетчик жестких ссылок)	Количество жестких ссылок на файл

Атрибут	Значение
UID	Идентификатор владельца файла
GID	Идентификатор группы, к которой принадлежит владелец файла
file size (размер файла)	Размер файла в байтах
last access time (время последнего доступа)	Время, когда к файлу последний раз производился доступ
last modify time (время последней модификации)	Время, когда файл последний раз модифицировался
last change time (время последнего изменения)	Время, когда последний раз изменялись права доступа к файлу, его UID, GID и значение счетчика жестких ссылок
inode number (номер индексного дескриптора)	Системный номер индексного дескриптора файла
file system ID (идентификатор файловой системы)	Идентификатор файловой системы, в которой находится файл

В ОС UNIX большая часть этой информации для любых файлов может быть получена с помощью команды *ls -l*.

Перечисленные атрибуты необходимы ядру для управления файлами. Например, когда пользователь пытается получить доступ к файлу, ядро сравнивает его UID и GID с UID и GID файла, чтобы определить, какая категория (пользователь, группа или прочие) разрешений на доступ должна быть задействована при установлении прав. Кроме того, время последней модификации файла используется в UNIX утилитой *make* для того, чтобы определить, какие исходные файлы новее, чем построенные на их основе выполняемые файлы, и требуют перекомпиляции.

Такого рода информацию можно получить для файлов всех типов, но не всегда следует трактовать ее буквально. В частности, атрибут "размер файла" не имеет никакого смысла для байт-ориентированных и блок-ориентированных файлов устройств.

В дополнение к указанным атрибутам в UNIX-системах для каждого файла устройства хранятся старший и младший номера устройства. В POSIX.1 поддержка файлов устройств зависит от реализации ОС, поэтому старший и младший номера устройства как стандартные атрибуты для них не определяются.

Атрибуты назначаются файлу ядром при его создании. Некоторые из них остаются неизменными в течение всего периода существования файла, а

некоторые могут изменяться при обращении к файлу. К атрибутам, которые для всех файлов остаются постоянными, относятся:

- тип файла;
- номер индексного дескриптора файла;
- идентификатор файловой системы;
- старший и младший номера устройства (для файлов устройств **только** в UNIX).

Остальные атрибуты изменяются следующими UNIX-командами и системными вызовами:

Команда UNIX	Системный вызов	Изменяемый атрибут
chmod	chmod	Право доступа и время последнего изменения
chown	chown	UID и время последней модификации
chgrp	chown	GID и время последней модификации
touch	utime	Время последнего доступа и время модификации
ln	link	Значение счетчика жестких ссылок (увеличивается)
rm	unlink	Значение счетчика жестких ссылок (уменьшается). Если значение этого счетчика равно нулю, файл удаляется из файловой системы
vi, emac	—	Размер файла, время последнего доступа и последней модификации

6.4. Индексные дескрипторы в UNIX System V

Рассмотрим назначение двух таких атрибутов файла, как номер индексного дескриптора и идентификатор файловой системы. Можно также заметить, что имена файлов не относятся к атрибутам, хранимым операционной системой для каждого файла. Используя в качестве примера UNIX System V, мы попробуем в этом разделе объяснить назначение этих атрибутов.

В UNIX System V для каждой файловой системы создается таблица индексных дескрипторов, в которой хранится информация обо всех файлах. Каждая запись в таблице индексных дескрипторов содержит все атрибуты файла, включая уникальный номер этого дескриптора и физический адрес на диске, где хранятся данные файла. Таким образом, если ядру необходим доступ к информации файла с номером индексного дескриптора, скажем, 15, то оно будет просматривать таблицу индексных дескрипторов и искать запись, содержащую индексный дескриптор с номером 15. Так как номер индексного дескриптора уникален только в пределах одной файловой системы, то запись индексного дескриптора файла определяется по идентификатору

файловой системы и номеру индексного дескриптора. Идентификатор присваивается файловой системе при выполнении команды *mount*, которая используется для получения доступа к файловым системам.

Операционная система не включает имя файла в запись индексного дескриптора, потому что установление соответствия между именами файлов и номерами индексных дескрипторов выполняется через каталоги. В частности, каталог содержит список имен и соответствующие им номера индексных дескрипторов для всех находящихся здесь файлов. Например, если каталог *foo* содержит файлы *xyz*, *a.out* и *xyz_ln1*, где *xyz_ln1* является жесткой ссылкой на *xyz*, то содержимое этого каталога будет выглядеть так, как показано на рис. 6.1 (большинство данных, зависящих от реализации ОС, опущено).

Для того чтобы получить доступ, например, к файлу */usr/joe*, ядро UNIX всегда может воспользоваться номером индексного дескриптора каталога "/" любого процесса (он сохраняется в U-области процесса и может быть изменен посредством системного вызова *chdir*). Ядро просматривает файл каталога "/" (используя запись для индексного дескриптора "/"), чтобы найти номер индексного дескриптора файла *usr*. Узнав номер дескриптора, оно проверяет, имеет ли вызывающий процесс право на поиск в каталоге *usr*, и получает доступ к содержимому файла *usr*. Затем ядро ищет номер индексного дескриптора файла *joe*.

Всякий раз, когда в каталоге создается новый файл, ядро UNIX создает новую запись в таблице индексных дескрипторов для сохранения информации о нем. Кроме того, ядро, присвоив этому файлу уникальный номер индексного дескриптора, добавляет его имя и номер дескриптора в соответствующий каталог.

Номер индексного дескриптора	Имя файла
115	.
89	..
201	xyz
346	a.out
201	xyz_ln1

Рис. 6.1. Примерное содержимое файла каталога

Номера индексных дескрипторов и идентификаторы файловых систем определены в POSIX.1, но использование этих атрибутов зависит от реализации ОС. Таблицы индексных дескрипторов содержатся в соответствующих файловых системах на диске, но ядро UNIX, чтобы иметь копию записей индексных дескрипторов, которые выбирались последними, ведет их таблицу и в оперативной памяти.

6.5. Интерфейсы прикладного программирования для файлов

В UNIX и в POSIX API для работы с файлами имеют ряд общих черт:

- Файлы идентифицируются путевыми именами.
- Прежде чем файлы можно будет использовать, их нужно создать. Ниже перечислены UNIX-команды и соответствующие им системные вызовы, предназначенные для создания различных типов файлов.

Тип файла	Команда UNIX	Системный вызов UNIX и POSIX.1
Обычные файлы	vi, ex и т.д.	open, creat
Каталоги	mkdir	mkdir, mknod
FIFO-файлы	mkfifo	mkfifo, mknod
Файлы устройств	mknod	mknod
Символические ссылки	ln -s	symlink

- Прежде чем прикладные программы смогут получить доступ к файлам, последние необходимо открыть. В UNIX и POSIX.1 определен API *open*, который может быть использован для открытия любого файла. Функция *open* возвращает дескриптор, который представляет собой идентификатор файла, предназначенный для использования в других системных вызовах, манипулирующих открытым файлом.
- Процесс может одновременно открыть максимум OPEN_MAX файлов любых типов. Значение OPEN_MAX и значение _POSIX_OPEN_MAX, определенные в POSIX.1, задаются в заголовке <limits.h>.
- Системные вызовы *read* и *write* могут использоваться для чтения данных из открытых файлов и записи данных в открытые файлы.
- Атрибуты файла можно запрашивать с помощью системного вызова *stat* или *fstat*.
- Атрибуты файла можно изменять посредством системных вызовов *chmod*, *chown*, *utime*, *link*.
- Жесткие ссылки на файл можно удалять с использованием системного вызова *unlink*.

Для облегчения запроса атрибутов файла прикладными программами в UNIX и POSIX.1 определен тип данных *struct stat* в заголовке <sys/stat.h>. Запись *struct stat* содержит все доступные для пользователя атрибуты любого запрашиваемого файла. Она инициализируется и возвращается функцией *stat* или *fstat*. Объявление *struct stat* согласно POSIX.1 выглядит так:


```
struct stat
{
  dev_t   st_dev;   /* идентификатор файловой системы */
  ino_t   st_ino;   /* номер индексного дескриптора файла */
  mode_t  st_mode;  /* тип файла и флаги доступа */
  nlink_t st_nlink; /* счетчик жестких ссылок */
  uid_t   st_uid;   /* UID владельца файла */
  gid_t   st_gid;   /* GID владельца файла */
  dev_t   st_rdev;  /* содержит старший и младший номера
                       устройства*/
  off_t   st_size;  /* размер файла в байтах */
  time_t  st_atime; /* время последнего доступа */
  time_t  st_mtime; /* время последней модификации */
  time_t  st_ctime; /* время последнего изменения статуса */
}
```

Если путевое имя (или дескриптор файла) символической ссылки передается как аргумент в системный вызов *stat* (или *fstat*), то эта функция произведет разыменование ссылки и покажет атрибуты реального файла, на который указывает ссылка. Для запроса атрибутов самой символической ссылки можно использовать системный вызов *lstat*. Поскольку символические ссылки стандартом POSIX.1 еще не поддерживаются, системный вызов *lstat* также не является для POSIX.1 стандартным.

6.6. Поддержка файлов ядром UNIX

В UNIX System V.3 ядро ведет таблицу файлов, в которой отслеживаются все открытые в системе файлы. Имеется также таблица индексных дескрипторов, содержащая копию индексных дескрипторов файлов, выбиравшихся последними.

Чтобы пользователь мог выполнить команду, ядро создает определенный процесс. Процесс имеет собственную структуру данных и, в частности, содержит таблицу дескрипторов файлов. В таблице дескрипторов имеется OPEN_MAX элементов, и в ней регистрируются все файлы, открытые процессом. Когда процесс вызывает функцию *open* с тем, чтобы открыть файл для чтения и (или) записи, ядро преобразует путевое имя в индексный дескриптор файла. Если индексный дескриптор файла не найден или если у процесса нет необходимых прав доступа к индексному дескриптору, вызов функции *open* завершается с кодом возврата -1, свидетельствующем об ошибке. Если индексный дескриптор файла доступен процессу, ядро продолжит организацию связи между записью в таблице дескрипторов файлов процесса (с помощью таблицы файлов ядра) и индексным дескриптором открываемого файла. Эта процедура выполняется в последовательности, описанной ниже:

1. Ядро ищет в таблице дескрипторов файлов процесса первую незадействованную позицию. Если такая позиция есть, она будет использована для обращения к файлу. Номер (индекс) этой позиции будет возвращен

процессу (через значение, возвращаемое функцией *open*) как дескриптор открытого файла.

2. Ядро просматривает таблицу файлов в своем пространстве и ищет незадействованную позицию, которую можно использовать для обращения к файлу. Если неиспользуемая позиция найдена, происходит следующее:
   а) в записи таблицы дескрипторов файлов процесса делается ссылка на найденную позицию в таблице файлов;
   б) в записи таблицы файлов производится ссылка на ту запись таблицы индексных дескрипторов файловой системы, в которой хранится индексный дескриптор этого файла;
   в) в записи таблицы файлов формируется указатель текущей позиции в открытом файле. Этот указатель представляет собой смещение относительно начала файла позиции, где будет происходить следующая операция чтения или записи;
   г) в запись таблицы файлов заносится информация о том, в каком режиме открыт файл: только для чтения, только для записи, для чтения и записи и т.д. Режим открытия задается вызывающим процессом как аргумент функции *open*;
   д) значение счетчика ссылок в записи таблицы файлов устанавливается равным 1. Этот базовый счетчик следит за тем, сколько дескрипторов файлов из процесса обращаются к данной записи;
   ж) значение счетчика ссылок индексного дескриптора файла в оперативной памяти увеличивается на 1. Счетчик определяет, сколько записей таблицы файлов указывает на этот индексный дескриптор.

Если условие (1) или (2) не выполнено, функция *open* возвратит значение -1 и ни одна запись таблицы дескрипторов файлов или таблицы файлов не будет предоставлена для нового файла.

На рис. 6.2 показаны таблица дескрипторов файлов процесса, таблица файлов ядра и таблица индексных дескрипторов файловой системы после того, как процесс открыл три файла: *xyz* — только для чтения, *abc* — для чтения и записи и вновь *abc* — только для записи.

Обратите внимание на то, что значение счетчика ссылок предоставленной записи таблицы файлов обычно равно 1, но процесс может с помощью функции *dup* (или *dup2*) сделать так, чтобы на одну запись таблицы файлов создавались ссылки из нескольких записей таблицы дескрипторов файлов. Как вариант, процесс может вызвать функцию *fork* и создать порожденный процесс, чтобы записи таблицы дескрипторов файлов порожденного и родительского процессов одновременно указывали на соответствующие записи таблицы файлов. Все это приводит к тому, что значение счетчика ссылок записи таблицы файлов превышает 1. Функции *dup*, *dup2*, *fork* и методы их использования будут объяснены более детально в главе 8.

Счетчик ссылок в записи индексного дескриптора файла показывает, сколько сделано ссылок из таблицы файлов на эту запись. Если значение

счетчика не равно нулю, это значит, что один или более процессов в текущий момент открывают файл для доступа.

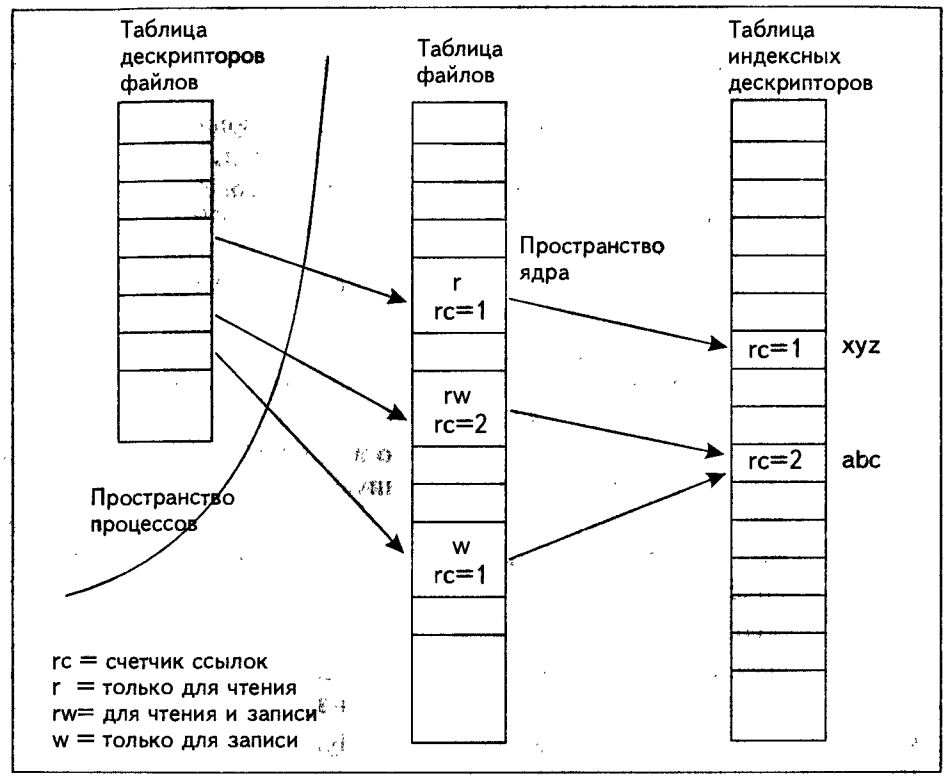


Рис. 6.2. Структура таблиц, предназначенных для манипулирования файлами

Если вызов функции *open* выполнен успешно, процесс может использовать возвращенный дескриптор файла для последующих обращений. В частности, когда процесс пытается читать данные из файла (или записывать их в файл), он использует дескриптор файла как первый аргумент системного вызова *read* (или *write*). Ядро будет использовать этот дескриптор файла как индекс в таблице дескрипторов файлов процесса для поиска элемента таблицы файлов, соответствующего открытому файлу. Затем ядро проверяет данные записи в таблице файлов, дабы убедиться в том, что файл открыт в соответствующем режиме, позволяющем выполнить необходимую операцию чтения или записи.

Если выяснилось, что операция чтения (или записи) совместима с режимом открытия файла, ядро использует указатель из записи в таблице файлов для доступа к записи индексного дескриптора файла (хранящегося в таблице индексных дескрипторов). Кроме того, оно использует указатель позиции в файле, хранящийся в записи таблицы файлов, чтобы определить, где должно происходить чтение или запись. Наконец, ядро проверяет тип

файла в записи индексного дескриптора и вызывает соответствующий драйвер для того, чтобы начать фактический обмен данными с физическим устройством.

Если процесс производит системный вызов *lseek* и изменяет указатель позиции в файле, задавая другое смещение для следующей операции чтения (или записи), ядро использует дескриптор файла как индекс в таблице дескрипторов файлов процесса при поиске соответствующей записи в таблице файлов. Затем ядро по записи в таблице файлов получает указатель на запись индексного дескриптора файла. Потом оно проверяет, не является ли этот файл байт-ориентированным файлом устройства, FIFO-файлом или символической ссылкой, так как такие файлы позволяют выполнять только последовательные операции чтения и записи. Если тип файла совместим с *lseek*, ядро изменит указатель позиции в файле, хранящийся в записи таблицы файлов, в соответствии со значением, указанным в аргументах функции *lseek*.

Когда процесс вызывает функцию *close*, чтобы закрыть открытый файл, выполняются следующие операции:

1. Ядро отмечает соответствующую позицию в таблице дескрипторов файлов данного процесса как незадействованную.
2. Ядро уменьшает значение счетчика ссылок соответствующей записи таблицы файлов на 1. Если значение счетчика ссылок все еще отлично от 0, выполняется операция, указанная в пункте 6.
3. Позиция таблицы файлов отмечается как незадействованная.
4. Значение счетчика ссылок в соответствующей записи таблицы индексных дескрипторов файла уменьшается на 1. Если значение счетчика ссылок все еще отлично от 0, выполняется операция, указанная в пункте 6.
5. Если значение счетчика жестких ссылок индексного дескриптора отлично от 0, ядро передает управление вызывающему процессу и возвращает код успешного выполнения. В противном случае ядро отмечает данную позицию таблицы индексных дескрипторов как незадействованную и освобождает всю физическую память, выделенную для хранения файла на диске, поскольку все путевые имена файла какой-то процесс удалил.
6. Ядро передает управление процессу и возвращает код успешного выполнения (0).

6.7. Взаимосвязь указателей потоков С и дескрипторов файлов

Указатели потоков С (FILE*) инициализируются путем вызова С-функции *fopen*. Указатель потока более эффективен при использовании в приложениях, осуществляющих крупномасштабные операции последовательного чтения из файлов и записи в файлы, поскольку библиотечные функции С при вводе-выводе в поток производят буферизацию. С другой стороны, дескриптор файла, выделенный системным вызовом *open*, является более

эффективным для приложений, часто выполняющих произвольную выборку данных из файлов, для осуществления которой буферизация ввода-вывода не желательна. Еще одно различие между этими функциями состоит в том, что указатели потоков поддерживаются всеми операционными системами, использующими компиляторы С и, в частности, VMS, CMS, DOS и UNIX. Дескрипторы файлов применяются только в системах, совместимых с UNIX и POSIX.1. Таким образом, программы, использующие указатели потоков, более мобильны, чем те, которые применяют дескрипторы файлов.

Для поддержки указателей потоков каждый процесс в UNIX имеет таблицу потоков фиксированного размера с OPEN_MAX количеством записей. Каждая запись имеет тип FILE и содержит рабочие данные для открытого файла. Данные, находящиеся в записи FILE, содержат информацию о буфере, выделенном для буферизации данных ввода-вывода, о состоянии флага ошибки ввода-вывода в файл, флаг конца файла и т.д. Функция *fopen* просматривает таблицу потоков вызывающего процесса, с тем чтобы найти неиспользованную позицию. Обнаруженная позиция служит для обозначения данного файла, а ее адрес (FILE*) является указателем потока. Кроме того, в UNIX функция *fopen* вызывает функцию *open* для фактического открытия файла, а запись FILE содержит дескриптор открытого файла. Дескриптор файла, связанный с указателем потока, можно извлечь с помощью макроса *fileno*, который объявляется в заголовке <stdio.h>:

```
int fileno (FILE* stream_pointer );
```

Таким образом, если процесс, чтобы открыть файл, вызывает функцию *fopen*, то возможность обращения к этому файлу реализуется путем выделения позиций в таблице потоков процесса и таблице дескрипторов файлов процесса. Если же для открытия файла вызывается функция *open*, то для обозначения файла назначается только позиция таблицы дескрипторов файлов процесса. Можно, однако, преобразовать дескриптор файла в указатель потока, воспользовавшись библиотечной функцией *C fdopen*:

```
FILE* fdopen (int file_descriptor, char* open_mode);
```

Действие функции *fdopen* подобно действию функции *fopen*, а именно: для обозначения файла она назначает позицию в таблице потоков процесса, регистрирует значение дескриптора в этой записи и возвращает адрес записи вызывающему процессу.

После вызова *fileno* или вызова *fdopen* процесс может обращаться к файлу как через указатель потока, так и через дескриптор файла. Другие библиотечные функции С при выполнении фактических операций с системными ресурсами тоже используют API операционной системы. Далее следует

список некоторых библиотечных функций C и соответствующих API ОС UNIX, которыми они пользуются при выполнении своих задач.

Библиотечные функции C	Используемые системные вызовы UNIX
fopen	open
fread, fgetc, fscanf, fgets	read
fwrite, fputc, fprintf, fputs	write
fseek, ftell, frewind	lseek
fclose	close

6.8. Каталоги

С точки зрения реализации файловой системы, каталог — это файл, содержащий записи. Каждая запись дает информацию о файле, находящемся в этом каталоге. Тип данных записи в UNIX System V и POSIX.1 — *struct dirent*, а в BSD UNIX — *struct direct*. Содержание записи зависит от реализации, но в системах UNIX и POSIX все они содержат два основных поля: имя файла и номер индексного дескриптора. Назначение каталогов — преобразование имен файлов в соответствующие индексные дескрипторы, с тем чтобы операционная система могла преобразовать любое путевое имя файла в номер индексного дескриптора и найти соответствующую ему запись.

Хотя любое приложение может использовать системные вызовы *open*, *read*, *write*, *lseek* и *close* для манипулирования каталогами, в UNIX и POSIX.1 определен набор переносимых функций, предназначенных специально для открытия, просмотра и закрытия каталога. Они создаются на основе системных вызовов *open*, *read*, *write* и *close* и определяются в заголовке <dirent.h> (в системах, совместимых с UNIX System V и POSIX.1) или в <sys/dir.h> (в BSD UNIX).

Функция	Назначение
opendir	Открывает каталог
readdir	Читает следующую запись из файла
closedir	Закрывает каталог
rewinddir	Устанавливает указатель позиции на начало файла

Функция *opendir* возвращает указатель типа DIR*, аналогичный указателю FILE* для потокового файла C. Этот указатель применяется в вызовах функций *readdir*, *rewinddir* и *closedir* для определения этого, каким каталогом следует манипулировать.

Кроме упомянутых в UNIX-системах определяются также функции *telldir* и *seekdir*, обеспечивающие произвольный доступ к различным записям каталога. Эти функции не относятся к стандарту POSIX.1 и аналогичны библиотечным C-функциям *ftell* и *fseek* соответственно.

Если процесс добавляет файл в каталог или удаляет его из каталога в то время, как другой процесс открыл файл посредством функции *opendir*, то в зависимости от реализации ОС этот другой процесс, применив функцию *readdir*, может увидеть, а может и не увидеть внесенные первым процессом изменения. Однако если последний процесс выполняет функцию *rewinddir* и затем читает каталог, используя функцию *readdir*, то, согласно POSIX.1, он должен прочитать самый последний вариант содержимого каталога.

6.9. Жесткие и символические ссылки

Жесткая ссылка — это путевое имя файла в ОС UNIX. Большинство UNIX-файлов имеют всего одну жесткую ссылку. Однако с помощью команды *ln* пользователи могут создавать дополнительные жесткие ссылки на файлы. Так, новую ссылку */usr/joe/book.new* для файла */usr/mary/fun.doc* может создать команда:

```
ln /usr/mary/fun.doc /usr/joe/book.new
```

Пользователи могут обращаться к такой ссылке как к файлу */usr/joe/book.new* или как к файлу */usr/mary/fun.doc*.

Символические ссылки могут создаваться тем же способом, что и жесткие ссылки, однако в команде *ln* необходимо указать опцию *-s*. С учетом сказанного вы можете создать */usr/joe/book.new* в качестве символической (а не жесткой) ссылки при помощи следующей команды:

```
ln -s /usr/mary/fun.doc /usr/joe/book.new
```

Символические и жесткие ссылки используются для обеспечения альтернативных способов обращения к файлам. Пусть, например, вы находитесь в каталоге */usr/jose/proj/doc* и постоянно просматриваете файл */usr/include/sys/unistd.h*. Вместо того чтобы при каждом обращении к этому файлу указывать полное путевое имя */usr/include/sys/unistd.h*, можно создать ссылку на него:

```
ln /usr/include/sys/unistd.h uniref
```

С этого момента вы можете обращаться к этому файлу как к *unref*. Таким образом, ссылки упрощают процесс обращения к файлам.

Команда *ln* отличается от команды *cp* тем, что последняя создает дубликат файла в другом файле и с другим путевым именем, тогда как *ln* прежде всего создает в каталоге новую запись, позволяющую обращаться к файлу. Например, после выполнения команды

```
ln /usr/jose/abc /usr/mary/xyz
```

каталоги */usr/jose* и */usr/mary* будут содержать следующее:

Номер индексного дескриптора	Имя файла
115	.
89	..
201	**abc**
346	a.out

/usr/jose

Номер индексного дескриптора	Имя файла
115	.
989	..
201	**xyz**
146	fun.c

/usr/mary

Обратите внимание на то, что и */usr/jose/abc*, и */usr/mary/xyz* ссылаются на один номер индексного дескриптора (201). Таким образом, никаких новых файлов не создано. Если, однако, мы воспользуемся командой *ln -s* или командой *cp* для создания файла */usr/mary/xyz*, то будет создан новый индексный дескриптор, а каталоги */usr/jose* и */usr/mary* будут выглядеть следующим образом:

Номер индексного дескриптора	Имя файла
115	.
89	..
201	**abc**
346	a.out

/usr/jose

Номер индексного дескриптора	Имя файла
115	.
989	..
345	**xyz**
146	fun.c

/usr/mary

Если файл */usr/mary/xyz* был создан с помощью команды *cp*, то его содержимое идентично содержимому файла */usr/jose/abc* и оба файла являются отдельными объектами в файловой системе. Однако если файл */usr/mary/xyz* был создан посредством команды *ln -s*, то он содержит только путевое имя */usr/mary/abc*.

Таким образом, команда *ln* позволяет экономить место на диске, так как не создает дубликатов файлов. Более того, всякий раз, когда пользователь вносит изменения в файл, они становятся доступными при обращении к этому файлу с использованием любой ссылки на него. Для файлов, созданных с помощью команды *cp*, это не характерно, поскольку дубликат и оригинал — это разные объекты.

Жесткие ссылки используются во всех версиях UNIX, но им присущ ряд ограничений:

- Пользователи не могут создавать жесткие ссылки на каталоги, не обладая правами привилегированного пользователя (root). Цель такого ограничения — воспрепятствовать созданию в файловой системе циклических ссылок. Вот пример циклической ссылки:

  ```
  ln /usr/jose/text/unix_link /usr/jose
  ```

  Если указанная команда выполнена успешно, то всякий раз, когда пользователь выполняет команду *ls -R /usr/jose*, она входит в бесконечный цикл рекурсивного отображения дерева подкаталогов каталога */usr/jose*. UNIX позволяет привилегированному пользователю создавать жесткие ссылки на каталоги, считая, очевидно, что уж он-то таких ошибок совершать не будет.
- Пользователи не могут создавать жесткие ссылки на файлы, находящиеся в другой файловой системе, так как жесткая ссылка — это лишь запись в файле каталога, которая обозначает тот же индексный дескриптор, что и исходная ссылка. Поскольку номера индексных дескрипторов уникальны только в пределах файловой системы, то возможна ситуация, когда в разных файловых системах присутствуют индексные дескрипторы с одинаковыми номерами.

Чтобы преодолеть упомянутые ограничения, в BSD UNIX используется система символических ссылок. Символическая ссылка позволяет обращаться к файлу в любой файловой системе, потому что содержимое ссылки — это путевое имя, а ядро операционной системы преобразует путевые имена и находит файлы и в локальной, и в удаленной файловых системах. Кроме того, пользователи имеют право создавать символические ссылки на каталоги, так как ядро может обнаруживать циклические ссылки на каталоги, создаваемые символическими ссылками. Поэтому никаких бесконечных циклов при просмотре каталогов не будет. Символические ссылки поддерживаются в UNIX System V.4, но не поддерживаются в POSIX.1.

Различия между символическими и жесткими ссылками описаны в таблице.

Жесткая ссылка	Символическая ссылка
Не создает нового индексного дескриптора	Создает новый индексный дескриптор
Не позволяет привилегированному пользователю ссылаться на каталоги	Может ссылаться на каталоги
Не может ссылаться на файлы в других файловых системах	Может ссылаться на файлы в других файловых системах
Увеличивает значение счетчика жестких ссылок соответствующего индексного дескриптора	Не изменяет значение счетчика жестких ссылок соответствующего индексного дескриптора

6.10. Заключение

В этой главе описаны различные типы файлов, используемых в файловых системах UNIX и POSIX. Показано, как создаются и применяются разные файлы, освещены общесистемные структуры данных и структуры данных процессов UNIX System V, которые предназначены для манипулирования файлами. Описаны интерфейсы прикладного программирования для файлов. Цель этой главы состоит в ознакомлении читателей с файловыми структурами UNIX, с тем чтобы они поняли, для чего были созданы системные вызовы UNIX и POSIX, как они работают и как могут применяться пользователями.

В следующей главе более подробно описываются API, предназначенные для работы с файлами UNIX и POSIX.

ГЛАВА 7

Файловые API операционной системы UNIX

Далее рассказывается о том, как приложения UNIX и POSIX взаимодействуют с файлами. Знакомство с этой главой позволит читателям научиться писать программы, выполняющие следующие функции для файлов любого типа в UNIX- и POSIX-системах:

- создание файлов;
- открытие файлов;
- пересылка данных в файлы и из файлов;
- закрытие файлов;
- удаление файлов;
- запрашивание атрибутов файлов;
- изменение атрибутов файлов;
- усечение файлов.

Применение файловых интерфейсов прикладного программирования (API) UNIX и POSIX.1 иллюстрируется на примерах программ, в которых реализуются UNIX-команды *ls*, *mv*, *chmod*, *chown* и *touch*, использующие эти API. Кроме того, в этой главе определяется класс языка C++, который называется *file*. Этот класс наследует все свойства класса *fstream* языка C++ и содержит ряд новых функций-членов, позволяющих создавать объекты для файлов любого типа, отображать и изменять атрибуты файловых объектов.

Предполагается, что читатели прочли предыдущую главу и ознакомились со структурами файлов UNIX и POSIX. Знание этого материала очень важно для понимания методики использования файловых API, описанных в настоящей главе.

7.1. Общие файловые API

Как разъяснялось в предыдущей главе, файлы в UNIX- и POSIX-системах могут относиться к следующим типам:

- обычный файл;
- каталог;
- FIFO-файл;
- байт-ориентированный файл устройства;
- блок-ориентированный файл устройства;
- символическая ссылка.

Для создания файлов существуют специальные API, которые будут описаны в следующих разделах. Ниже представлен набор базовых API, с помощью которых можно манипулировать файлами различных типов.

API	Назначение
open	Открывает файл для доступа к данным
read	Читает данные из файла
write	Записывает данные в файл
lseek	Обеспечивает произвольную выборку данных из файла
close	Закрывает соединение с файлом
stat, fstat	Запрашивает атрибуты файла
chmod	Изменяет права доступа к файлу
chown	Изменяет UID и (или) GID файла
utime	Изменяет дату и время последнего изменения содержимого файла и последнего доступа к нему
link	Создает жесткую ссылку на файл
unlink	Удаляет жесткую ссылку на файл
umask	Устанавливает маску

Далее все перечисленные общие API описываются более детально.

7.1.1. Функция open

Функция *open* устанавливает соединение между процессом (процесс — это прикладная программа, находящаяся в стадии выполнения) и файлом. С помощью этой функции можно создавать совершенно новые файлы. Любой процесс, вызвав функцию *open*, может получить дескриптор для обращения к вновь созданному файлу. Дескриптор файла используется в системных вызовах *read* и *write* для получения доступа к его содержимому.

Прототип функции *open* выглядит так:

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

int open ( const char *path_name, int access_mode, mode_t permission );
```

Первый аргумент, *path_name*,— это имя файла. Им может быть абсолютное путевое имя (символьная строка, начинающаяся символом "/") или относительное путевое имя (символьная строка, не начинающаяся символом "/"). Если *path_name* — символическая ссылка, то рассматриваемая функция преобразует ее (рекурсивно, если символическая ссылка обращается к другой символической ссылке) в имя файла, который упоминается в данной ссылке.

Второй аргумент, *access_mode*,— это целое значение, которое показывает, какие виды доступа к файлу разрешены вызывающему процессу. Значение *access_mode* должно быть одним из макросов, определенных в заголовке <fcntl.h>.

Флаг режима доступа	Использование
O_RDONLY	Открывает файл только для чтения
O_WRONLY	Открывает файл только для записи
O_RDWR	Открывает файл для чтения и записи

Кроме того, можно задать один или несколько из перечисленных ниже модификаторов, логически складывая их поразрядно с одним из указанных выше флагов режима доступа и изменяя таким образом механизм доступа к файлу.

Флаг модификатора доступа	Использование
O_APPEND	Добавляет данные в конец файла
O_CREAT	Создает файл, если он не существует
O_EXCL	Используется только с флагом O_CREAT. Если установлены флаги O_CREAT и O_EXCL, а указанный файл уже существует, выполнение функции *open* завершается неудачей
O_TRUNC	Если файл существует, отбрасывает его содержимое и устанавливает размер файла равным 0
O_NONBLOCK	Указывает на то, что все последующие операции чтения из файла и записи в файл должны выполняться без блокировки
O_NOCTTY	Говорит о том, что указанный файл терминального устройства нельзя использовать как управляющий терминал вызывающего процесса

Проиллюстрируем использование этих флагов на примере оператора, который открывает файл */usr/xyz/textbook* для чтения и записи в режиме добавления:

```
int fdesc = open ( "/usr/xyz/textbook",O_RDWR | O_APPEND, 0 );
```

Если файл необходимо открыть только для чтения, он должен уже существовать и никакие другие флаги модификаторов использовать нельзя.

Если файл открывается только для записи или для чтения и записи, можно указывать любые флаги модификаторов. При этом флаги O_APPEND, O_TRUNC, O_CREAT и O_EXCL применимы только к обычным файлам, O_NONBLOCK — только к FIFO-файлам и файлам устройств, а O_NOCTTY — только к файлам терминальных устройств.

Флаг O_APPEND указывает на то, что данные будут добавляться в конец файла. Если этот флаг отсутствует, данные могут записываться в любое место файла.

Флаг O_TRUNC определяет, что если заданный файл уже существует, функция *open* должна отбросить его содержимое. Если этот флаг отсутствует, текущие данные файла функцией *open* не изменяются.

Флаг O_CREAT говорит о том, что если заданный файл не существует, функция *open* должна создать его. Если же заданный файл существует, этот флаг на функцию *open* никак не влияет. Если заданный файл не существует и файл O_CREAT не указан, выполнение функции *open* прерывается и возвращается код неудачного завершения. Файл O_EXCL, если он используется, должен сопровождаться флагом O_CREAT. Если указаны оба эти флага и заданный файл существует, то попытка выполнить функцию *open* завершается неудачей. Таким образом, флаг O_EXCL гарантирует, что в результате вызова функции *open* будет создан новый файл.

Флаг O_NONBLOCK указывает на то, что если вызов функции *open* и все последующие вызовы функций *read* и *write* для заданного файла заблокируют вызывающий процесс, ядро должно немедленно прервать выполнение этих функций и возвратить управление процессу с соответствующим кодом состояния. Процесс, как правило, блокируется при попытках чтения пустого канала (функция *pipe* описывается в разделе 7.5) и записи в заполненный канал. С помощью этого флага можно указать, что такие операции чтения и записи не являются блокирующими. В UNIX System V.3 вместо этого флага определен флаг O_NDELAY; по своему назначению они похожи, но работают по-разному. Более подробно эти флаги описаны в разделе 7.5.

Флаг O_NOCTTY определен в стандарте POSIX.1. Он указывает на то, что если процесс не имеет управляющего терминала и открывает файл терминального устройства, то этот терминал не будет управляющим терминалом процесса. Если названный флаг не установлен, то решение вопроса о том, будет ли этот терминал управляющим терминалом процесса, зависит от реализации ОС. Отметим, что в UNIX System V.3, где флаг O_NOCTTY не определен, вызов функции *open* при отсутствии у процесса управляющего терминала автоматически назначает управляющим терминалом первый открытый файл терминального устройства.

Аргумент *permission* необходим только в том случае, если в аргументе *access_mode* установлен флаг O_CREAT. Он задает права доступа к файлу для его владельца, членов группы и всех остальных пользователей. В UNIX System V (V.3 и ниже) этот аргумент имеет тип *int* и его значение обычно задается как восьмеричный целочисленный литерал, например 764. Левая, средняя и правая цифры восьмеричного числа задают права доступа соответственно для владельца, группы и прочих пользователей. В каждой восьмеричной цифре левый, средний и правый биты определяют соответственно права на чтение, запись и выполнение. Значение каждого бита может быть равно либо 1 (право доступа предоставляется), либо 0 (право доступа не предоставляется). Так, число 764 означает, что у владельца нового файла есть право на чтение, запись и выполнение, у членов группы — на чтение и запись, у прочих пользователей — только на чтение.

Тип данных аргумента *permission* определяется в POSIX.1 как *mode_t*, а значение его должно строиться на основании макросов, определенных в заголовке <sys/stat.h>. Эти макросы являются псевдонимами восьмеричных целых значений, используемых в UNIX System V. Например, значение 764 должно быть задано так:

```
S_IRWXU | S_IRGRP | S_IWGRP | S_IROTH
```

Значение *permission*, указанное в вызове функции *open*, модифицируется значением *umask* вызывающего процесса. Значение *umask* задает те права доступа, которые необходимо автоматически маскировать (исключать) для всех файлов, создаваемых процессом. Значение *umask* наследуется процессом от его родительского процесса и может запрашиваться и изменяться системным вызовом *umask*. Вот прототип функции *umask*:

```
mode_t umask ( mode_t new_umask );
```

В качестве аргумента функция *umask* принимает новое значение маски. С момента начала выполнения этой функции вызывающий процесс использует заданное новое значение, а функция возвращает старое. Например, следующий оператор присваивает текущее значение маски переменной *old_mask* и устанавливает новое значение маски как "отсутствие права выполнения для группы" и "отсутствие права записи и выполнения для прочих пользователей":

```
mode_t old_mask = umask ( S_IXGRP | S_IWOTH | S_IXOTH );
```

Функция *open* получает значение аргумента *permission* и логически складывает его поразрядно со значением, равным дополнению до единицы значения *umask* вызывающего процесса. Таким образом, для каждого вновь создаваемого файла назначается следующее разрешение на доступ:

```
actual_permission = permission & ~umask_value
```

Следовательно, биты, составляющие значение *umask*, показывают, что у всех вновь создаваемых файлов соответствующие права доступа будут исключаться. Например, если *open* вызывается в UNIX System V.3 для создания файла */usr/mary/show_ex* с правами доступа 0557, а значение *umask* вызывающего процесса — 031, то для этого файла назначаются следующие виды доступа:

```
actual_permission = 0557 & (~031) = 0546
```

Функция *open* возвращает значение -1, если данный API выполнен неудачно и переменная *errno* содержит код ошибки. Если API выполнен успешно, то возвращается дескриптор файла, по которому к файлу можно обращаться в других системных вызовах. Значение дескриптора файла должно принадлежать диапазону от 0 до OPEN_MAX-1 включительно.

7.1.2. Системный вызов creat

Системный вызов *creat* служит для создания обычных файлов. Вот его прототип:

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

int creat ( const char* path_name, mode_t mode );
```

Аргумент *path_name* — это путевое имя создаваемого файла. Аргумент *mode* анологичен применяемому в API *open*. Поскольку к данному интерфейсу был добавлен флаг O_CREAT, его можно использовать и для создания обычных файлов, и для их открытия. Поэтому API *creat* устарел и сохраняется только для обратной совместимости с ранними версиями ОС UNIX. Функцию *creat* можно реализовать с помощью функции *open*:

```
#define creat(path_name,mode)
  open (path_name, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

7.1.3. Функция read

Функция *read* выбирает блок данных фиксированного размера из файла с указанным дескриптором. Прототип функции *read* выглядит следующим образом:

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

ssize_t read ( int fdesc, void* buf, size_t size );
```


Первый аргумент, *fdesc*, — это целочисленный дескриптор, обозначающий открытый файл. Второй аргумент, *buf*, — адрес буфера, содержащего прочитанные данные. Третий аргумент, *size*, задает количество байтов, которое необходимо прочитать из файла. Тип данных *size_t* определяется в заголовке <sys/types.h> и должен быть таким же, как и тип данных *unsigned int*.

Следует отметить, что функция *read* может читать текстовые и двоичные файлы. Именно по этой причине тип данных аргумента *buf* — это универсальный указатель (*void**). Например, приведенный ниже фрагмент кода последовательно читает одну или более записей данных типа *struct sample* из файла *dbase*:

```
struct sample { int x; double y; char* z; } varX;
int fd = open ("dbase", O_RDONLY);
while (read(fd,&varX,sizeof(varX))>0)
  /* данные процесса записаны в varX */
```

Функция *read* возвращает значение, указывающее количество байтов данных, успешно прочитанных и сохраненных в аргументе *buf*. Как правило, оно должно быть равно значению *size*. Если же файл содержит менее *size* байтов непрочитанных данных, то возвращаемое этой функцией значение будет меньше *size*. Если встречается признак конца файла, *read* возвращает нулевое значение.

Поскольку *ssize_t* обычно определяется в заголовке <sys/types.h> как *int*, пользователи не должны в вызове функции *read* задавать значение *size*, превышающее INT_MAX. При соблюдении этого требования значение, возвращаемое функцией, всегда будет соответствовать числу фактически прочитанных байтов.

Для случая, когда вызов функции *read* прерывается перехваченным сигналом (сигналы рассматриваются в главе 9) и операционная система не перезапускает этот вызов автоматически, в POSIX.1 предусмотрено два варианта поведения функции *read*. Первый вариант — тот же, что в UNIX System V.3, где *read* возвращает значение -1, *errno* устанавливается в EINTR и все данные, прочитанные в этом вызове, отбрасываются (следовательно, процесс не может их восстановить). Второй вариант задан стандартом POSIX.1 FIPS: функция *read* возвращает число байтов данных, прочитанных до прерывания. Это позволяет процессу продолжать чтение файла.

В BSD UNIX, где ядро автоматически перезапускает системный вызов после его прерывания сигналом, *read* возвращает такое же значение, как и при нормальном выполнении. В UNIX System V.4 пользователь может для каждого сигнала указать, должно ли ядро перезапускать прерываемый системный вызов. Таким образом, с перезапускаемыми вызовами функция *read* может вести себя так, как в BSD UNIX, а с неперезапускаемыми — так, как в UNIX System V.3 или POSIX.1 FIPS.

Функция *read* может блокировать выполнение вызывающего процесса, если она читает FIFO-файл или файл устройства, а данных для удовлетворения запроса на чтение еще нет. Для запроса неблокирующих операций чтения содержимого конкретного файла пользователь может указать с

дескриптором файла флаг O_NONBLOCK или O_NDELAY. Поведение функции *read* с такими особыми файлами будет подробно описано в разделах, посвященных API FIFO-файлов и файлов устройств.

7.1.4. Функция write

Функция *write* помещает блок данных фиксированного размера в файл, обозначенный соответствующим дескриптором. Ее действие противоположно действию функции *read*. Прототип этой функции выглядит следующим образом:

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

ssize_t write ( int fdesc, const void* buf, size_t size );
```

Первый аргумент, *fdesc*, — это целочисленный дескриптор, обозначающий открытый файл. Второй аргумент, *buf*,— адрес буфера, содержащего данные, которые предназначены для записи в файл. Третий аргумент, *size*, задает количество байтов, находящихся в аргументе *buf*.

Как и API *read*, API *write* может работать с текстовыми и двоичными файлами. Именно по этой причине тип данных аргумента *buf* — универсальный указатель (*void**). Например, представленный ниже фрагмент кода записывает десять записей данных типа *struct sample* в файл *dbase2*:

```
struct sample { int x; double y; char* z; } varX[10];
int fd = open ("dbase2", O_WRONLY);

/* здесь инициализировать массив varX... */

write (fd, (void*)varX, sizeof varX );
```

Функция *write* возвращает количество байтов данных, успешно записанных в файл. Как правило, оно должно быть равно значению *size*. Если же в результате выполнения *write* размер файла превысит установленный системой лимит или если диск файловой системы переполнится, то функция *write* возвратит число байтов, фактически записанных до прекращения выполнения.

Прерывания, вызванные сигналами, функцией *write* обрабатываются так же, как функцией *read*. Если в процессе выполнения функции *write* поступает какой-то сигнал и операционная система не перезапускает этот системный вызов автоматически, *write* может либо выдать значение -1 и установить для *errno* значение EINTR (как это делается в System V), либо выдать число байтов данных, записанных до прерывания сигналом. Последний вариант оговорен как обязательный стандартом POSIX.1 FIPS.

Как и для функции *read*, в UNIX System V.4 пользователь может для каждого сигнала указать, должно ли ядро перезапускать прерываемый этим

сигналом системный вызов. Таким образом, с перезапускаемыми вызовами функция *write* может вести себя так, как в BSD UNIX (и возвращать такое же значение, как при нормальном выполнении), а с неперезапускаемыми — так, как в UNIX System V.3 или POSIX.1 FIPS.

И, наконец, функция *write* может выполнять неблокирующую операцию, если для аргумента *fdesc* установлен флаг O_NONBLOCK или O_NDELAY. В этом она идентична функции *read*.

7.1.5. Функция close

Функция *close* отсоединяет файл от процесса. Прототип этой функции представлен ниже:

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

Аргумент *fdesc* — это целочисленный дескриптор, обозначающий открытый файл. В случае успешного выполнения функция *close* возвращает 0, в случае неудачи возвращает -1, а *errno* устанавливается в код ошибки.

Функция *close* освобождает неиспользуемые дескрипторы файлов, чтобы их можно было задействовать для обозначения других файлов. Это весьма важно, поскольку процесс одновременно может открыть до OPEN_MAX файлов, а функция close позволяет ему, многократно используя дескрипторы файлов, обращаться в ходе выполнения к большему, чем OPEN_MAX, числу файлов.

Кроме того, функция *close* освобождает ресурсы системы (например, элементы таблицы файлов и буфер, выделенные для хранения прочитанных из файла и записанных в буфер данных), которые предназначены для поддержки функционирования дескрипторов файлов. Это сокращает потребность процесса в памяти.

Если процесс завершается, не закрыв все открытые файлы, ядро само их закрывает.

В классе *iostream* определяется функция-член *close*, которая закрывает файл, связанный с объектом потока ввода-вывода. Эта функция-член может быть реализована с помощью API *close* следующим образом:

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

int iostream::close() {return close(this->fileno());}
```

7.1.6. Функция fcntl

Функция *fcntl* помогает пользователю запрашивать и устанавливать флаги управления доступом и флаг *close-on-exec* любого дескриптора файла. Кроме

того, с помощью этой функции пользователи для обозначения одного файла могут назначать несколько дескрипторов. Прототип этой функции:

```
#include <fcntl.h>
int fcntl ( int fdesc, int cmd, ... );
```

Аргумент *cmd* задает операции, которые необходимо выполнить над файлом, обозначенным аргументом *fdesc*. Третий аргумент, который может быть указан после *cmd*, зависит от значения *cmd*. Возможные значения аргумента *cmd*, определенные в заголовке <fcntl.h>, перечислены ниже.

Значение *cmd*	Использование
F_GETFL	Возвращает флаги управления доступом дескриптора файла *fdesc*
F_SETFL	Устанавливает и сбрасывает флаги управления доступом, которые заданы в третьем аргументе функции *fcntl*. Возможные флаги управления доступом — O_APPEND и O_NONBLOCK (в POSIX-совместимых UNIX-системах — O_NDELAY)
F_GETFD	Возвращает флаг *close-on-exec* файла, обозначенного аргументом *fdesc*. Если возвращаемое значение равно 0, то флаг сбрасывается; в противном случае возвращается ненулевое значение и флаг устанавливается. По умолчанию для вновь открываемого файла флаг *close-on-exec* сброшен
F_SETFD	Устанавливает и сбрасывает флаг *close-on-exec* дескриптора файла *fdesc*. Третий аргумент функции *fcntl* — целочисленное значение. При сбросе флага оно должно быть равно 0, при установке — 1
F_DUPFD	Дублирует дескриптор файла *fdesc* другим дескриптором. Третий аргумент функции *fcntl* — целочисленное значение, свидетельствующее о том, что дубликат дескриптора должен быть больше этого значения или равен ему. В данном случае *fcntl* возвращает дескриптор-дубликат

Функция *fcntl* используется для изменения флага управления доступом к дескриптору файла. Например, если файл открыт для блокирующего доступа в режиме чтения и записи, а процессу нужно изменить режим доступа на неблокирующую запись с добавлением, он может вызвать *fcntl* с дескриптором этого файла:

```
int cur_flags = fcntl(fdesc, F_GETFL);
int rc = fcntl(fdesc, F_SETFL, cur_flag | O_APPEND | O_NONBLOCK);
```

Флаг *close-on-exec* дескриптора файла означает, что если процесс-владелец данного дескриптора вызывает API *exec* для выполнения другой программы, этот дескриптор должен быть закрыт ядром до запуска новой программы (если флаг установлен) или не должен (если флаг сброшен). Более подробно API *exec* и флаг *close-on-exec* рассматриваются в главе 8. В приведенном ниже

примере программа сообщает состояние флага *close-on-exec* дескриптора файла *fdesc* и устанавливает его:

```
cout << fdesc << "close-on-exec:" << fcntl(fdesc, F_GETFD) << endl;
(void)fcntl(fdesc, F_SETFD, 1);  // установить флаг close-on-exec
```

С помощью функции *fcntl* можно дублировать дескриптор файла *fdesc* другим дескриптором. В результате получаются два дескриптора, обозначающих один и тот же файл, с одинаковым режимом доступа (чтение и/или запись, блокирующий или неблокирующий и т.д.), совместно использующих для чтения и записи файла один указатель. Это полезно при переназначении стандартного ввода и вывода на файл. Например, следующие операторы переназначают стандартный ввод процесса на файл с именем *FOO*:

```
int fdesc = open("FOO", O_RDONLY);  // открыть FOO для чтения
close (0);                          // закрыть стандартный ввод
if (fcntl(fdesc,F_DUPFD, 0)==-1) perror("fcntl");
                                    // с этого момента стандартный
                                    // ввод осуществляется из файла
FOO
char buf[256];
int rc = read(0,buf,26);            // читать данные из FOO
```

UNIX-функции *dup* и *dup2* выполняют ту же задачу дублирования файловых дескрипторов, что и *fcntl*. Их можно реализовать с помощью *fcntl* следующим образом:

```
#define dup(fdesc)          fcntl(fdesc,F_DUPFD,0)
#define  dup2(fdesc1, fd2)  close(fd2), fcntl(fdesc,F_DUPFD,fd2)
```

Функция *dup* дублирует дескриптор файла *fdesc* наименьшим из неиспользуемых дескрипторов вызывающего процесса. Функция *dup2* дублирует дескриптор файла *fdesc* с помощью дескриптора *fd2* независимо от того, используется он для обозначения другого файла или нет.

Дублирование файлов и переназначение стандартного ввода-вывода более подробно описаны в следующей главе.

Значение, возвращаемое функцией *fcntl*, зависит от значения *cmd*, но в случае неудачи оно всегда равно -1. Неудачное выполнение может быть связано с ошибками, возникающими при задании *fdesc* и *cmd*.

7.1.7. Функция lseek

Системные вызовы *read* и *write* всегда выполняются относительно текущей позиции указателя чтения-записи в файле. С помощью системного вызова *lseek* значение текущей позиции можно изменить. Таким образом, функция *lseek* позволяет процессу произвольно выбирать данные из любого открытого файла. При этом она неприменима к FIFO-файлам, байт-ориентированным файлам устройств и символическим ссылкам.

Прототип этой функции:

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

off_t lseek ( int fdesc, off_t pos, int whence );
```

Первый аргумент, *fdesc*, — это целочисленный дескриптор, обозначающий открытый файл. Второй аргумент, *pos*, задает смещение в байтах, которое должно быть прибавлено к базовому адресу для получения нового значения смещения. Базовый адрес задается аргументом *whence*, который может иметь одно из следующих значений:

Значение *whence*	Базовый адрес
SEEK_CUR	Текущий адрес указателя в файле
SEEK_SET	Начало файла
SEEK_END	Конец файла

Значения SEEK_CUR, SEEK_SET и SEEK_END определяются в заголовке <unistd.h>. Следует помнить, что если значение *whence* равно SEEK_SET, то задавать отрицательное значение *pos* не разрешается, так как функция в этом случае назначит отрицательное смещение в файле. Если выполнение вызова *lseek* дает в результате новое смещение, выходящее за текущий признак конца файла, возможны два варианта: если файл открыт только для чтения, *lseek* выдает сообщение об ошибке; если файл открыт для записи, *lseek* выполняется успешно и увеличивает размер файла до значения, равного SEEK_SET. Данные, находящиеся между концом файла и новым относительным адресом, инициализируются символами NULL.

Функция *lseek* возвращает новый относительный адрес, по которому будет выполняться следующая операция чтения или записи. В случае неудачи возвращается -1.

В классе *iostream* определены функции *tellg* и *seekg*, позволяющие произвольно выбирать данные из любого объекта потока ввода-вывода. Их можно реализовать с помощью API *lseek*. Это будет выглядеть так:

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

streampos iostream::tellg()
{
   return (streampos)lseek(this->fileno(), (off_t)0, SEEK_CUR );
}

iostream&iostream::seekg(streampos pos, seek_dir ref_loc )
{
```



```
   if (ref_loc == ios::beg )
      (void)lseek(this->fileno(), (off_t)pos, SEEK_SET);
   else if ( ref_loc == ios::cur )
      (void)lseek(this->fileno(), (off_t)pos, SEEK_CUR);
   else if ( ref_loc == ios::end )
      (void)lseek(this->fileno(), (off_t)pos, SEEK_END);
   return *this;
}
```

Функция *iostream::tellg* дает *lseek* указание возвратить текущий указатель на файл, связанный с объектом потока ввода-вывода. Файловый дескриптор объекта потока ввода-вывода выдается функцией-членом *fileno*. Отметим, что *streampos* и *off_t* являются данными того же типа, что и *long*.

Функция *iostream::seekg* смещает с помощью *lseek* указатель в файле, связанном с объектом потока ввода-вывода. Аргументы *iostream::seekg* — это смещение в файле и базовый адрес. Между значениями *seek_dir* и значениями *whence*, которые используются функцией *lseek*, существует полное соответствие.

Значение *seek_dir*	Значение *whence*
ios::beg	SEEK_SET
ios::cur	SEEK_CUR
ios::end	SEEK_END

Таким образом, функция *iostream::seekg* просто преобразует значение *seek_dir* в значение *lseek_whence* и с помощью вызова *lseek* смещает указатель в файловом объекте потока ввода-вывода в соответствии со значением *pos*. Файловый дескриптор потока ввода-вывода возвращается функцией-членом *fileno*.

7.1.8. Функция link

Функция *link* создает новую ссылку на существующий файл. Новый файл при этом не создается, а формируется лишь новое путевое имя для уже существующего файла.

Вот прототип функции *link*:

```
#include <unistd.h>

int link ( const char* cur_link, const char* new_link );
```

Первый аргумент, *cur_link*, — это путевое имя существующего файла. Второй аргумент, *new_link*, — это новое путевое имя, которое должно быть присвоено этому же файлу. Если вызов выполняется успешно, то счетчик жестких ссылок файла увеличивается на единицу.

В ОС UNIX с помощью функции *link* нельзя создавать жесткие ссылки, указывающие на файлы, которые находятся в других файловых системах. Более того, *link* можно использовать с каталогами только в том случае, если она вызывается процессом, имеющим права привилегированного пользователя.

В ОС UNIX с помощью API *link* реализована команда *ln*. Вот простая версия программы *ln*, которая не поддерживает опцию *-s* (создание символической ссылки):

```
/* test_ln.C */
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>

int main ( int argc, char* argv[] )
{
    if ( argc!=3 )    {
      cerr << "usage: " << argv[0] << " <src_file> <dest_file>\n";
      return 0;
   }
    if ( link ( argv[1], argv[2] ) == -1 ) {
      perror( "link" );
      return 1;
   }
   return 0;
}
```

7.1.9. Функция unlink

Функция *unlink* удаляет ссылку на существующий файл. Эта функция уменьшает значение счетчика жестких ссылок указанного файла и удаляет соответствующую запись из каталога. Если функция выполнена успешно, то к данному файлу уже нельзя будет обращаться с помощью этой ссылки. Файл удаляется из файловой системы, когда счетчик его жестких ссылок становится равным нулю и ни один процесс не использует дескриптор, обозначающий этот файл.

Вот прототип функции *unlink*:

```
#include <unistd.h>

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

Аргумент *cur_link* — это путевое имя существующего файла. Если вызов успешен, возвращается 0; в противном случае возвращается -1. Неудачный вызов *unlink* может быть обусловлен неправильным заданием аргумента (файла с таким именем нет), отсутствием у вызывающего процесса разрешения на удаление этого путевого имени и, наконец, прерыванием функции каким-либо сигналом.

В ОС UNIX функция *unlink* может удалить каталог только в том случае, если вызывающему процессу назначены права привилегированного пользователя.

В ANSI C определена функция *rename*, которая выполняет операцию, аналогичную *unlink*. Если аргументом функции *rename* служит пустой каталог, она удаляет его.

Прототип функции *rename* имеет следующий вид:

```
#include <stdio.h>

int rename ( const char* old_path_name, const char* new_path_name );
```

И функция *link*, и функция *rename* возвращают код ошибочного завершения, если создаваемая ссылка и исходный файл находятся в разных файловых системах (или в разных разделах диска).

В ОС UNIX с помощью API *link* и *unlink* можно реализовать команду *mv*. Ниже приведена простая версия программы *mv*:

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

int main ( int argc, char* argv[] )
{
   if ( argc!=3 )|||!strcmp(argv[1], argv[2]) )
     cerr << "usage: " << argv[0] << " <old_link> <new_link>\n";
   else if (link (argv[1], argv[2]) == 0)
     return unlink(argv[1]);
   return -1;
}
```

В командной строке этой программы даны два аргумента: *old_link* и *new_link*. Сначала программа проверяет, являются ли эти аргументы разными путевыми именами; в этом случае программа просто завершает свою работу, поскольку изменять нечего. Затем программа вызывает функцию *link* для создания *new_link* как новой ссылки на *old_link*. Если функция *link* не выполняется, программа выдает код возврата -1; в противном случае она вызывает *unlink* для удаления *old_link* и возвращает значение, равное коду возврата функции *unlink*.

7.1.10. Функции stat и fstat

Функции *stat* и *fstat* возвращают атрибуты заданного файла. Различие между ними состоит в том, что первый аргумент *stat* — это путевое имя файла, тогда как первый аргумент *fstat* является дескриптором файла. Прототипы функций *stat* и *fstat* имеют вид:

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

int stat (const char* path_name, struct stat* statv);
int fstat (const int fdesc, struct stat* statv);
```

Второй аргумент обеих рассматриваемых функций — это адрес переменной типа *struct stat*. Тип данных *struct stat* определен в заголовке <sys/stat.h>. Объявление этой структуры, одинаковое для UNIX и POSIX.1, выглядит следующим образом:

```
struct stat
{
    dev_ts  t_dev;      /* идентификатор файловой системы */
    ino_t   st_ino;     /* номер индексного дескриптора файла */
    mode_t  st_mode;    /* содержит тип файла и флаги доступа */
    nlink_t st_nlink;   /* значение счетчика жестких ссылок */
    uid_t   st_uid;     /* идентификатор владельца файла */
    gid_t   st_gid;     /* идентификатор группы */
    dev_t   st_rdev;    /* содержит старший и младший номера устройства */
    off_t   st_size;    /* размер файла в байтах */
    time_t  st_atime;   /* время последнего доступа */
    time_t  st_mtime;   /* время последней модификации */
    time_t  st_ctime;   /* время последнего изменения статуса */
];
```

Если вызов *stat* или *fstat* успешен, возвращается 0; в противном случае возвращается -1. Неудача вызова *stat* и *fstat* может быть обусловлена неправильным заданием путевого имени файла (в *stat*) или дескриптора (в *fstat*), отсутствием у вызывающего процесса разрешения на доступ к файлу, а также прерыванием функции каким-либо сигналом.

Если в качестве путевого имени в *stat* задана символическая ссылка, *stat* преобразует ее и обращается к несимволической ссылке. Так же себя ведет, как вы помните, API *open*. Следовательно, *stat* и *fstat* нельзя использовать для получения атрибутов самих символических ссылок. Для решения этой проблемы в BSD UNIX имеется API *lstat*. Прототип этой функции такой же, как у *stat*:

```
int lstat (const char* path_name, struct stat* statv);
```

Функция *lstat* ведет себя точно так же, как *stat* с файлами-несимволическими ссылками. Если же аргумент *path_name* в *lstat* представляет собой символическую ссылку, данная функция возвращает атрибуты этой символической ссылки, а не того файла, на который она указывает. Функция *lstat* используется также в UNIX System V.3 и V.4, но в POSIX.1 не определена.

На базе API *stat* реализована UNIX-команда *ls*. Применив эту команду с опцией *-l*, можно, в частности, вывести на экран данные типа *struct stat* для указанного файла. Приведенная ниже программа *test_ls.C* эмулирует UNIX-команду *ls -l*:

```
/* Program to emulate the UNIX ls -l command */
#include <iostream.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>
static char xtbl[10] = "rwxrwxrwx";

#ifndef MAJOR
#define MINOR_BITS      8
#define MAJOR(dev)  ((unsigned]dev >> MINOR_BITS)
#define MINOR(dev) ( dev & MINOR_BITS)
#endif

/* показать тип файла в столбце 1 выходной строки */
static void display_file_type ( ostream& ofs, int st_mode )
{
   switch (st_mode&S_IFMT] {
       case S_IFDIR: ofs << 'd'; return; /* каталог */
       case S_IFCHR: ofs << 'c'; return; /* байт-ориентированный файл
                                            устройства */
       case S_IFBLK: ofs << 'b'; return; /* блок-ориентированный файл
                                            устройства */
       case S_IFREG: ofs << '-'; return; /* обычный файл */
       case S_IFLNK: ofs << 'l'; return; /* символическая ссылка */
       case S_IFIFO: ofs << 'p'; return; /* FIFO-файл */
   ]
]

/* показать права доступа для владельца, группы и прочих
   пользователей, а также все специальные флаги */
 static void display_access_perm ( ostream& ofs, int st_mode )
{
   char amode[10];
    for (int i=0, j=(1 << 8); i<9; i++, j>>=1)
      amode[i]=(st_mode&j)?xtbl[i]:'-';  /* установить права доступа */
   if (st_mode&S_ISUID) amode[2] = (amode[2]=='x') ? 'S' : 's';
   if (st_mode&S_ISGID) amode[5] = (amode[5]=='x') ? 'G' : 'g';
   if (st_mode&S_ISVTX) amode[8] = (amode[8]=='x') ? 'T' : 't';
   ofs << amode << ' ';
}

/* перечислить атрибуты одного файла */
static void long_list (ostream& ofs, char* path_name)
{
    struct stat    statv;
    struct group   *gr_p;
    struct passwd  *pw_p;
```

```
  if (stat(path_name,&statv))
  {
     perror( path_name );
     return;
   }
   display_file_type( ofs, statv.st_mode );
   display_access_perm( ofs, statv.st_mode );
   ofs << statv.st_nlink;            /* отобразить значение счетчика
                                        жестких ссылок */
   gr_p = getgrgid(statv.st_gid);    /* преобразовать GID в имя группы */
   pw_p = getpwuid(statv.st_uid);    /* преобразовать UID в имя
                                        пользователя */
   ofs << ' ' << pw_p->pw_name << ' ' << gr_p->gr_name << ' ';
   if ((statv.st_mode&S_IFMT) == S_IFCHR ||
        (statv.st_mode&S_IFMT)==S_IFBLK)
     ofs << MAJOR(statv.st_rdev) << ',' << MINOR(statv.st_rdev);
  else ofs << statv.st_size;        /* показать размер файла или
                                        старший и младший номера
                                        устройства */

   ofs << ' ' << ctime(&statv.st_mtime); /* показать время последней
                                            модификации */
   ofs << ' ' << path_name << endl;      /* показать имя файла */
}

/* главный цикл отображения атрибутов для каждого файла */
int main (int argc, char* argv[])
{
   if (argc==1)
     cerr << "usage: " << argv[0] << " path name>...\n";
   else while (--argc = 1) long_list( cout, *++argv);
   return 0;
}
```

Эта программа принимает в качестве аргументов одно или несколько путевых имен файлов. Для каждого путевого имени она вызывает функцию *long_list* с целью отображения атрибутов соответствующего файла в формате UNIX-команды *ls -l*. Атрибуты каждого файла выводятся в одной физической строке, которая организована следующим образом:

- первое поле (столбец 1): односимвольный код, описывающий тип файла;
- второе поле (столбцы 2—4): права доступа для чтения, записи и выполнения, предоставляемые владельцу файла;
- третье поле (столбцы 5—7): права доступа для чтения, записи и выполнения, предоставляемые группе;
- четвертое поле (столбцы 8—10): права доступа для чтения, записи и выполнения, предоставляемые прочим пользователям;
- пятое поле: значение счетчика жестких ссылок на файл;
- шестое поле: имя владельца файла;
- седьмое поле: имя группы, владеющей файлом;

- восьмое поле: размер файла в байтах или старший и младший номера устройства, если это файл байт-ориентированного или блок-ориентированного устройства;
- девятое поле: дата последней модификации файла;
- десятое поле: имя файла.

Переменная *st_mode* в записи *struct stat* содержит несколько атрибутов: тип файла, права доступа владельца, права доступа группы, права доступа прочих пользователей, флаг смены идентификатора пользователя (*set-UID*), флаг смены идентификатора группы (*set-GID*) и бит-липучку (*sticky bit*). В заголовке <sys/stat.h> определены макросы, с помощью которых можно извлекать все эти поля, как показано в приведенной выше программе.

При кодировании типа файла одним символом соблюдаются правила, принятые в UNIX-команде *ls -l*: *d* обозначает каталог, *c* — байт-ориентированный файл устройства, *b* — блок-ориентированный файл устройства, "-" — обычный файл, *p* — FIFO-файл, *l* — символическую ссылку.

Права доступа для всех категорий пользователей всегда указываются в следующем порядке: чтение, запись, выполнение. Эти виды доступа обозначаются соответственно буквами *r*, *w* и *x*. Дефис на месте одной из этих букв означает, что соответствующий вид доступа для данной категории пользователей запрещен.

Флаги *set-UID*, *set-GID* и *sticky*-бит характерны для UNIX. Если флаг *set-UID* исполняемого файла установлен, то эффективный идентификатор пользователя всех процессов, создаваемых в результате выполнения этого файла, будет совпадать с идентификатором владельца файла. Так, если идентификатор владельца файла равен 0 (в UNIX это идентификатор привилегированного пользователя), у соответствующего процесса будут права привилегированного пользователя. Точно так же, если у файла установлен флаг *set-GID*, то эффективный идентификатор группы всех процессов, создаваемых в результате выполнения этого файла, будет совпадать с идентификатором группы. Эффективный идентификатор пользователя и эффективный идентификатор группы процесса служат для определения прав доступа процесса к любому файлу. В частности, ядро сначала сверяет эффективный идентификатор пользователя, от имени которого выполняется процесс, с идентификатором владельца файла. Если они совпадают, процессу для доступа к файлу предоставляются права владельца файла. Если эффективный идентификатор процесса отличается от идентификатора владельца файла, но эффективный идентификатор группы процесса совпадает с идентификатором группы, владеющей файлом, процесс получает права доступа, установленные для группы. В случае полного несоответствия предоставляются права доступа по категории "прочие пользователи".

Флаги *set-UID* и *set-GID* оказываются весьма полезными в некоторых UNIX-программах, например, в программе *passwd*. Для выполнения этих программ процессам нужны права привилегированного пользователя (например, чтобы программа *passwd* могла поменять пользовательский пароль

путем исправления соответствующей записи в файле */etc/passwd* или */etc/shadow*). Сменив эффективный идентификатор пользователя при выполнении этих программ, пользователи получают такой же результат, как если бы они обладали правами привилегированного пользователя.

Эффективный идентификатор пользователя и эффективный идентификатор группы применяются также в тех случаях, когда процесс создает файл. Идентификатор владельца файла устанавливается равным эффективному идентификатору пользователя, от имени которого выполняется процесс, а идентификатор группы файла назначается в зависимости от системы: в UNIX System V.3 идентификатор группы файла устанавливается равным эффективному идентификатору группы процесса, тогда как в BSD UNIX идентификатор группы файла устанавливается равным идентификатору группы каталога, который содержит этот файл. Стандарт POSIX.1 допускает применение обоих этих методов назначения идентификатора группы файла. В UNIX System V.4 идентификатор группы нового файла устанавливается равным идентификатору группы каталога, который содержит этот файл (метод BSD), если флаг *set-GID* этого каталога установлен. В противном случае файлу назначается тот же идентификатор группы, который имеет создавший его процесс (метод System V.3).

Если для исполняемого файла установлен *sticky*-бит, то после завершения процесса текст программы (код команд) остается в оперативной памяти. Благодаря этому в следующий раз при выполнении программы ядро сможет запустить этот процесс быстрее. Этот флаг зарезервирован для часто выполняемых программ, например для shell ОС UNIX и редактора vi. Устанавливать и сбрасывать для файлов *sticky*-бит может только привилегированный пользователь.

Имя владельца и имя группы, которой принадлежит файл, поддерживаются в ОС UNIX, но стандартом POSIX как обязательные не оговариваются. Функция *getpwuid* преобразует идентификатор пользователя в имя пользователя, а функция *getgrgid* преобразует идентификатор группы в имя группы. Эти функции определены соответственно в заголовках <pwd.h> и <grp.h>.

Размер файла указывается для файлов, каталогов и именованных каналов. Для файлов устройств функция *long_list* выводит на экран старший и младший номера устройства, которые извлекаются из поля *st_rdev* записи *struct stat*. Для обеспечения быстрого доступа к этим числам в некоторых UNIX-системах используются макросы MAJOR и MINOR (они определяются в заголовке <sys/stat.h>). Если они системой не определены, то наша программа-пример определяет их явно. MINOR_BITS — это число младших битов в поле *st_rdev*, используемых для хранения младшего номера устройства (в большинстве UNIX-систем используется 8 битов), а остальные биты поля *st_rdev* служат для хранения старшего номера устройства.

Последние два поля хранят информацию о дате последней модификации и имени файла. Они извлекаются соответственно из поля *st_mtime* двух аргументов *statv* и *path_name*.

Ниже даны результаты выполнения представленной нами программы:

```
% a.out /etc/motd /dev/fd0 /usr/bin
-rw-r-xrwx   1  joe     unix     25    July 5, 1997    /etc/motd
crw-r--r-x   2  mary    admin    30    June 25, 1997   /dev/fd0
drwxr-xr--   1  terry   sharp    15    Oct. 16, 1996   /usr/bin
```

7.1.11. Функция access

Функция *access* проверяет факт существования указанного файла и (или) права доступа, предоставляемые пользователю. Прототип этой функции выглядит следующим образом:

```
#include <unistd.h>

int access ( const char* path_name, int flag );
```

Аргумент *path_name* — это путевое имя файла. Аргумент *flag* содержит один или несколько битовых флагов, которые определены в заголовке <unistd.h>:

Битовый флаг	Назначение
F_OK	Проверяет факт существования указанного файла
R_OK	Проверяет, есть ли у вызывающего процесса право на чтение
W_OK	Проверяет, есть ли у вызывающего процесса право на запись
X_OK	Проверяет, есть ли у вызывающего процесса право на выполнение

Значение аргумента *flag* вызова *access* формируется путем побитового логического сложения одного или нескольких из этих битовых флагов. Например, следующий оператор проверяет, есть ли у пользователя права на чтение и запись файла */usr/foo/access.doc*:

```
int rc = access("/usr/foo/access.doc",R_OK|W_OK);
```

Если значение *flag* равно *F_OK*, то функция возвращает 0 при наличии указанного файла, в противном случае код возврата равен -1.

Если значение *flag* представляет собой любое сочетание *R_OK*, *W_OK* и *X_OK*, то функция *access* сверяет реальные идентификаторы пользователя и группы, с правами которых выполняется процесс, с идентификаторами владельца и группы указанного файла. При этом функция *access* определяет соответствующую категорию доступа (владелец, группа, прочие) и для каждой категории проверяются значения битовых флагов *flag*. Если все затребованные права доступа предоставляются, функция возвращает 0. В противном случае возвращается -1.

В приведенной ниже программе *test_access.C* функция *access* используется для того, чтобы определить, существует ли указанный файл. Это делается для

каждого аргумента командной строки. Если указанного файла не существует, он создается и инициализируется символьной строкой "*Hello world*". Если файл существует, программа просто читает из него данные:

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

int main (int argc, char* argv[])
{
   char  buf[256];
   int   fdesc, len;
   while ( --argc>0)
   {
      if (access(*++argv,F_OK))    // новый файл
      {
        fdesc = open(*argv,O_WRONLY|O_CREAT,0744);
        write(fdesc,"Hello world\n",12);
      } else                       // файл существует, прочитать данные
      {
         fdesc = open(*argv,O_RDONLY);
         while (len=read(fdesc,buf,256))
            write(fdesc,buf,len);
      }
      close(fdesc);
    } /* для каждого аргумента командной строки */
   return 0;
}
```

Эту несложную программу можно использовать как основу для построения программы управления базой данных. Если нужно будет создать новую базу данных, то программа будет инициализировать ее файл некоторыми начальными данными бухгалтерского учета; если файл базы данных уже существует, программа будет читать из него некоторую начальную информацию, чтобы проверить совместимость программы и файла базы данных по версиям и т.д.

7.1.12. Функции chmod, fchmod

Функции *chmod* и *fchmod* изменяют права доступа к файлу для владельца, группы и прочих пользователей, а также флаги *set-UID*, *set-GID* и sticky-бит. Процесс, который вызывает одну из этих функций, должен иметь эффективный идентификатор — либо привилегированного пользователя, либо владельца файла. На базе API *chmod* реализованы UNIX-команды *chmod*.

Прототипы функций *chmod* и *fchmod* выглядят следующим образом:

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

int chmod ( const char* path_name, mode_t flag );
int fchmod ( int fdesc, mode_t flag );
```

Аргумент *path_name* функции *chmod* — это путевое имя файла, а аргумент *fdesc* функции *fchmod* — дескриптор файла. Аргумент *flag* содержит новые права доступа и специальные флаги, которые необходимо установить для данного файла. Значение *flag* соответствует таковому в API *open*; его можно задавать либо как восьмеричное целочисленное, либо строить из макросов, определенных в заголовке <sys/stat.h>. Например, следующая функция устанавливает флаг *set-UID*, отменяет право на запись для категории "группа" и право на чтение и выполнение для категории "прочие" для файла */usr/joe/funny.book*:

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

void change_mode()
{
    int flag = (S_IWGRP | S_IROTH | S_IXOTH);
    struct stat statv;
    if (stat("/usr/joe/funny.book",&statv))
      perror("stat");
    else
    {
      flag = (statv.st_mode & ~flag) | S_ISUID;
      if (chmod ("usr/joe/funny.book", flag))
      perror("chmod");
    }
}
```

В этой программе сначала вызывается функция *stat*, которая выявляет текущие права доступа к файлу */usr/joe/funny.book*. Затем в *statv.st_mode* с помощью битовой маски отменяются право на запись для категории "группа" и право на чтение и выполнение для категории "прочие". После этого в *statv.st_mode* устанавливается флаг *set-UID*. Все остальные флаги не модифицируются. Полученный в результате аргумент *flag* передается в функцию *chmod* для внесения указанных изменений. Если вызов *chmod* или *stat* неудачен, программа вызывает функцию *perror*, которая выдает диагностическое сообщение.

В отличие от API *open*, в функции *chmod* права доступа, указанные в аргументе *flag*, значением *umask* вызывающего процесса не модифицируются.

7.1.13. Функции chown, fchown, lchown

Функции *chown* и *fchown* изменяют идентификатор владельца и идентификатор группы указанного файла. Различаются они только первым аргументом, который обозначает файл (т.е. путевым именем или дескриптором). На базе этих API реализованы UNIX-команды *chown* и *chgrp*. Функция *lchown* похожа на функцию *chown*, правда, имеется одно отличие: если аргумент *path_name* — это символическая ссылка на какой-либо файл, то функция *lchown* изменяет принадлежность файла-символической ссылки, тогда как функция *chown* изменяет принадлежность файла, на который эта ссылка указывает.

Прототипы этих функций выглядят следующим образом:

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

int chown ( const char* path_name, uid_t uid, gid_t gid );
int fchown ( int fdesc, uid_t uid, gid_t gid );
int lchown ( const char* path_name, uid_t uid, gid_t gid );
```

Аргумент *path_name* — это путевое имя файла. Аргумент *uid* задает новый идентификатор владельца файла. Аргумент *gid* задает новый идентификатор группы, который должен быть присвоен файлу. Если фактическое значение *uid* или *gid* равно -1, то соответствующий идентификатор файла не изменяется.

В BSD UNIX менять идентификатор владельца и идентификатор группы с помощью рассматриваемых функций может только процесс, обладающий правами привилегированного пользователя. Если, однако, эффективный идентификатор процесса совпадает с идентификатором владельца файла, а эффективный идентификатор группы или один из идентификаторов дополнительных групп, в которые входит владелец процесса, совпадает с идентификатором группы файла, то процесс может менять только идентификатор группы файла.

В UNIX System V менять идентификатор пользователя и идентификатор группы файла может только процесс, чей эффективный идентификатор пользователя совпадает либо с идентификатором владельца файла, либо с идентификатором привилегированного пользователя.

В POSIX.1 установлено, что если переменная _POSIX_CHOWN_RESTRICTED определена со значением, отличным от -1, то функция *chown* должна вести себя так же, как в BSD UNIX. Если же эта переменная не определена, то *chown* должна вести себя, как в UNIX System V.

Если *chown* вызывается процессом, который не имеет прав привилегированного пользователя, и успешно выполняется, изменяя владельца какого-либо файла, то флаги *set-UID* и *set-GID* этого файла сбрасываются. Это не позволяет пользователям создавать программы, которые можно выполнять от лица других пользователей (например, root), чтобы затем пользоваться их привилегиями.

Если *chown* вызывается процессом с эффективным идентификатором привилегированного пользователя, то трактовка функцией *chown* флагов *set-UID* и *set-GID* файлов, которые она модифицирует, зависит от реализации. В UNIX System V.3 и V.4 эти флаги остаются без изменений.

UNIX-программа *chown* реализована с помощью следующей программы *test_chown.C*:

```
#include <iostream.h>
#include <stdio.h>
#include <sys/types.h
#include <sys/stat.h>
#include <unistd.h>
#include <pwd.h>
int main (int argc, char* argv[])
{
  if (argc 3)
  {
    cerr << "Usage: " << argv[0] << <usr_name> <file> ... .\n";
    return 1;
   ]
   struct passwd * pwd  = getpwnam(argv[1]);/* преобразовать имя
                                               пользователя в UID */
   uid_t  UID  = pwd->pw_uid;
   struct stat  statv;
   if (UID == (uid_t)-1)
     cerr<< "Invalid user name\n";
   else for (int i=2; i < argc; i++)      /* для каждого из указанных
                                             файлов */
   if (stat(argv[i],&statv)==0)
   {
     if (chown(argv[i], UID, statv.st_gid)) perror("chown");}
else perror("stat");
   return 0;
}
```

Данная программа принимает как минимум два аргумента командной строки: первый — это имя нового владельца файла, второй и последующие аргументы — путевые имена файлов. Программа сначала преобразует заданное имя пользователя в идентификатор пользователя (с помощью функции *getpwuid*). Если преобразование выполнено успешно, программа обрабатывает каждый из указанных файлов следующим образом: вызывает *stat* для получения идентификатора группы файла, затем вызывает *chown* для изменения идентификатора владельца файла. Если *stat* или *chown* дает сбой, вызывается функция *perror*, которая выводит на экран диагностическое сообщение.

7.1.14. Функция utime

Функция *utime* изменяет дату и время последнего доступа и модификации файла. Прототип этой функции выглядит следующим образом:

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

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

Аргумент *path_name* — это путевое имя файла. Аргумент *times* задает новое время обращения к файлу и новое время его модификации. Структура *utimbuf* определяется в заголовке <utime.h>:

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

В POSIX.1 *struct utimbuf* определяется в заголовке <utime.h>, а в UNIX System V — в заголовке <sys/types.h>. Тип данных *time_t* соответствует типу данных *unsigned long*, а сами данные — это количество секунд, прошедших со дня "рождения" UNIX, т.е. с полудня 1 января 1970 г. по универсальному времени.

Если аргумент *times* задан как 0, этот API устанавливает время доступа и время модификации указанного файла равными текущему времени. При этом вызывающий процесс должен иметь право на запись в указанный файл, а эффективный идентификатор его владельца должен совпадать либо с идентификатором владельца файла, либо с идентификатором привилегированного пользователя.

Если *times* — адрес переменной типа *struct utimbuf*, API устанавливает время доступа и время модификации файла в соответствии со значениями, заданными в этой переменной. При этом эффективный идентификатор владельца вызывающего процесса должен совпадать либо с идентификатором пользователя файла, либо с идентификатором привилегированного пользователя.

В случае успешного выполнения *utime* возвращает 0, а в случае неудачи — -1. Причинами неудачи могут быть ошибки, допущенные при задании аргумента *path_name*, отсутствие у процесса прав доступа к указанному файлу, разные владельцы файла и процесса, неверный адрес аргумента *times*.

В приведенной ниже программе *test_touch.C* с помощью функции *utime* изменяются время доступа и время модификации файлов.

```
/* Emulate the UNIX touch program */
#include <iostream.h> 630SO
#include <stdio.h>
```



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

int main (int argc, char* argv[])
{
    struct utimbuf  times;
    times.actime = times.modtime = time(0);
    while (--argc  0)            /* обработать все указанные файлы */
      if (utime (*++argv, &times)) perror("utime");
    return 0;
}
```

Эта программа определяет переменную *times* типа *struct utimbuf* и инициализирует ее значением текущего времени (полученного из вызова функции *time*). Последующими аргументами командной строки должны быть одно или несколько путевых имен файлов. Для каждого из них программа вызывает функцию *utime*, которая обновляет время доступа и время модификации.

7.2. Блокировка файлов и записей

UNIX-системы позволяют множеству процессов одновременно читать и модифицировать один и тот же файл, что дает им возможность совместно использовать данные. Но при этом возникают и определенные сложности, так как процессу трудно установить, когда данные, находящиеся в интересующем его файле, используются другим процессом. Этот момент особенно важен для таких приложений, как управление базами данных, где ни один процесс не может записывать данные в файл или читать их из файла, к которому в текущий момент осуществляет доступ другой процесс. Чтобы решить эту проблему, в UNIX- и POSIX-системах поддерживается механизм блокировки файлов. Блокировка может применяться только при использовании обычных файлов. Этот механизм позволяет процессу заблокировать файл так, чтобы другие процессы не могли модифицировать его до того, как он будет разблокирован первым процессом.

В частности, процесс может установить блокировку записи или блокировку чтения на часть файла или на весь файл. Разница между блокировкой записи и чтения состоит в том, что блокировка записи не позволяет другим процессам устанавливать на блокированную часть файла свои блокировки чтения или записи. Блокировка чтения не позволяет другим процессам устанавливать на эту часть файла только свои блокировки записи. Блокировку чтения эти процессы устанавливать могут. Таким образом, назначение блокировки записи — не давать другим процессам читать заблокированную область файла и записывать в нее данные в течение того времени, когда процесс, который установил блокировку, модифицирует эту область. Блокировку записи называют также *исключающей блокировкой*. Назначение блокировки чтения — не давать другим процессам записывать данные в заблокированную область в то время, как процесс, который установил

блокировку, читает из нее данные. Другим процессам разрешается читать данные из заблокированных областей и блокировать их повторно. Поэтому блокировка чтения называется также *разделяемой блокировкой*.

Следует подчеркнуть, что блокировки, введенные ядром операционной системы, являются обязательными для всех процессов. Если на файл установлена обязательная исключающая блокировка, ни один процесс не может с помощью системных вызовов *read* и *write* обращаться к данным в заблокированной области. Точно так же, если на область файла установлена обязательная разделяемая блокировка, ни один процесс не может с помощью системного вызова *write* изменять данные в заблокированной области. С помощью этих механизмов можно синхронизировать чтение и запись совместно используемых файлов несколькими процессами: если один процесс блокировал файл, то другие процессы, которые хотят записать данные в заблокированные области, блокируются до тех пор, пока первый не снимет блокировку. Обязательные блокировки, однако, могут вызывать проблемы. В частности, если "заблудший" процесс установил обязательную исключающую блокировку файла и не снимает ее, другие процессы не могут получить доступ к этой области файла до тех пор, пока "заблудший" процесс не будет уничтожен или пока система не будет перезагружена. UNIX System V.3 и V.4 поддерживают обязательные блокировки, а BSD UNIX и POSIX-системы — нет.

Если блокировка файла не является обязательной, она считается рекомендуемой. Рекомендуемая блокировка не контролируется ядром на уровне системных вызовов. То есть несмотря на то, что для файла установлена блокировка чтения или записи, другие процессы все равно могут обращаться к нему через API *read* и *write*. Для обеспечения возможности использовать рекомендуемые блокировки процессы, которые манипулируют одним и тем же файлом, должны согласовать свою работу так, чтобы в каждой операции чтения и записи соблюдалась следующая последовательность:

- производится попытка блокировать область, к которой будет осуществляться доступ. В случае неудачи процесс может либо подождать, пока запрос блокировки не будет удовлетворен, либо перейти к выполнению других операций и попробовать вновь блокировать файл позже;
- после блокировки производится чтение заблокированной области или запись в нее;
- блокировка снимается.

Процесс, стараясь при каждой операции установить рекомендуемую блокировку на область файла, с которой он собирается работать, не нарушает этим никакой блокирующей защиты, установленной для этой области другими процессами; другие же процессы не будут модифицировать эту область, пока блокировка не будет снята. После выполнения операции процесс должен сразу же снять все блокировки, которые он установил на файл, чтобы открыть доступ к его ранее заблокированным областям. Рекомендуемая блокировка считается безопасной, поскольку никакой "заблудший" процесс не сможет принудительно блокировать какой-либо файл, и остальные процессы

могут продолжать работу, читать данные из файла или записывать данные в файл после фиксированного числа неудачных попыток блокировки.

Недостаток рекомендуемых блокировок состоит в том, что программы, которые создают процессы, осуществляющие совместное использование файлов, должны следовать описанной выше процедуре блокировки файлов, иначе их действия будут несогласованными. Это условие может оказаться трудным для контроля, если программы получены из разных источников (например, от разных поставщиков программного обеспечения). Все UNIX- и POSIX-системы поддерживают рекомендуемые блокировки.

Для блокировки файлов в UNIX System V и POSIX.1 используется API *fcntl*. В частности, с помощью этого API можно устанавливать блокировки чтения и записи как на весь файл, так и на отдельный его фрагмент. В BSD UNIX 4.2 и 4.3 интерфейс *fcntl* возможность блокировки файлов не поддерживает. Прототип API *fcntl* выглядит следующим образом:

```
#include <fcntl.h>

int fcntl ( int fdesc, int cmd_flag, ... );
```

Аргумент *fdesc* — это дескриптор файла, подлежащего обработке. Аргумент *cmd_flag* определяет, какую операцию необходимо выполнить. Возможные значения этого аргумента определены в заголовке <fcntl.h>.

cmd_flag	**Назначение**
F_SETLK	Блокирует файл. Не блокирует, если эта операция сразу не завершается успешно
F_SETLKW	Блокирует файл и блокирует вызывающий процесс на все время, пока действует блокировка на файл
F_GETLK	Делает запрос о том, какой процесс заблокировал указанную область файла

При блокировке файлов третьим аргументом функции *fcntl* является адрес переменной типа *struct flock*. Эта переменная задает область файла, где нужно установить, снять блокировку или запросить ее состояние. Структура *flock* объявляется в <fcntl> следующим образом:

```
struct flock
{
    short l_type;   /* какую блокировку нужно установить или снять */
    short l_whence; /* базовый адрес следующего поля */
    off_t l_start;  /* смещение относительно базового адреса l_whence */
    off_t l_len;    /* сколько байтов находится в заблокированной области */
    pid_t l_pid;    /* PID процесса, который блокировал файл */
};
```

Поле *l_type* задает тип устанавливаемой или снимаемой блокировки. Возможные значения этого поля, которые определены в заголовке <fcntl.h>, и соответствующие им функции указаны ниже:

Значение *l_type*	Использование
F_RDLCK	Устанавливает на указанную область блокировку чтения (разделяемая)
F_WRLCK	Устанавливает на указанную область блокировку записи (исключающая)
F_UNLCK	Снимает блокировку с указанной области

Поле *l_len* задает размер заблокированной области начиная с начального адреса, определенного полями *l_whence* и *l_start*. Если значение *l_len* — положительное число, то это число является выраженной в байтах длиной заблокированной области. Если *l_len* равно 0, то заблокированная область занимает диапазон от начального адреса до заданного системой лимита на максимальный размер файла. Это значит, что при увеличении размера файла действие блокировки распространяется на его расширенную область. Параметр *l_len* не может иметь отрицательного значения.

Переменная типа *struct flock* определяется и устанавливается процессом до передачи в вызов *fcntl*. Если аргумент *cmd_arg* вызова *fcntl* равен F_SETLK или F_SETLKW, данная переменная определяет область файла, подлежащую блокированию или разблокированию. Если *cmd_arg* = F_GETLK, то значение этой переменной используется и в качестве входных данных, и как возвращаемая переменная. В частности, при возвращении из *fcntl* эта переменная содержит размер и адрес заблокированной области файла и идентификатор процесса, заблокировавшего область. Возвращаемый идентификатор процесса находится в поле *l_pid* этой переменной.

Если процесс блокирует файл для чтения, например, начиная с адреса 0 и по адрес 256, а затем блокирует этот же файл для записи с адреса 0 по адрес 512, то процесс будет продолжать блокировать только записи с адреса 0 по адрес 512. Ранее установленная блокировка чтения с 0 по 256 будет перекрыта блокировкой записи. Этот механизм называется *продвижением блокировки*. Если процесс теперь разблокирует область файла с 128 по 480, то ему будут принадлежать две блокировки записи: одна с 0 по 127, а другая с 481 по 512. Этот механизм называется *разделением блокировки*.

Блокировка, установленная функцией *fcntl*, является рекомендуемой. Обязательные блокировки в POSIX.1 не поддерживаются. UNIX System V.3 и V.4 в отличие от POSIX позволяют устанавливать с помощью *fcntl* обязательные блокировки. Чтобы осуществить это, необходимо сначала задать такие атрибуты файла как установление флага *set-GID* и предоставление права на выполнение для категории "группа", после чего все блокировки, определяемые функцией *fcntl* для этого файла, получат статус.

Чтобы показать, что устанавливаемые на файл блокировки чтения и записи являются обязательными, в UNIX System V.3 и V.4 можно использовать команду *chmod*. Эта команда имеет следующий синтаксис:

```
chmod a+l <имя_файла>
```

Все, установленные процессом блокировки файлов при его завершении будут сняты. Если процесс блокирует файл, а затем с помощью функции *fork* (см. следующую главу) создает порожденный процесс, то последний не наследует блокировку файла, установленную его родителем.

В результате успешного выполнения функция *fcntl* возвращает 0, а в случае неудачи возвращает -1. Среди возможных причин неудачи следует назвать ошибку при задании дескриптора файла, конфликт между блокируемой или освобождаемой областью и блокировками, установленными другим процессом, наличие неверных данных в третьем аргументе, достижение установленного системой лимита максимального числа блокировок записей на один файл.

Возможность использования функции *fcntl* для блокировки файла показана ниже:

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

int main (int argc, char* argv[])
 {
   struct flock  fvar;
   int           fdesc;
   while (--argc > 0) {  /* выполнить следующее для каждого файла */
      if ((fdesc=open(*++argv,O_RDWR))==-1)
      {
          perror("open"); continue;
      }
      fvar.l_type = F_WRLCK;
      fvar.l_whence = SEEK_SET;
      fvar.l_start = 0;
      fvar.l_len = 0;
      /* попытка установки исключающей блокировки (блокировки записи)
         на весь файл */
       while (fcntl(fdesc, F_SETLK,&fvar)==-1)
      {
         /* попытка блокировки неудачна, выяснить, кто заблокировал файл */
         while (fcntl(fdesc,F_GETLK,&fvar)!=-1 &&
             fvar.l_type!=F_UNLCK)
          {
             cout << *argv << " locked by " << fvar.l_pid
                  << " from " << fvar.l_start << " for "
                  << fvar.l_len << " byte for " <<
                  (fvar.l_type==F_WRLCK ? 'w' : 'r') << endl;
             if (!fvar.l_len) break;
             fvar.l_start += fvar.l_len;
```

```
            fvar.l_len = 0;
          } /* пока есть блокировки, установленные другими процессами */
        } /* пока попытка блокирования не будет успешной */
        /* файл заблокирован; теперь обработать данные в файле */
        /* ... */
        /* теперь снять блокировку со всего файла */
        fvar.l_type = F_UNLCK;
        fvar.l_whence = SEEK_SET;
        fvar.l_start = 0;
        fvar.l_len = 0;
        if (fcntl(fdesc, F_SETLKW,&fvar)==-1)  perror("fcntl")
    }
    return 0;
}   /* main */
```

В качестве аргументов в этой программе используется одно или несколько путевых имен. Для каждого из указанных файлов программа пытается с помощью функции *fcntl* установить рекомендуемую блокировку. Если вызов *fcntl* неудачен, программа просматривает файл и выводит информацию обо всех блокировках на стандартный вывод. В частности, для каждой заблокированной области программа сообщает:

- путевое имя файла;
- идентификатор процесса, который заблокировал область;
- начальный адрес заблокированной области;
- длину заблокированной области;
- вид блокировки: исключающая (блокировка записи) или разделяемая (блокировка чтения).

Цикл выполняется до тех пор, пока вызов функции *fcntl* не будет выполнен успешно. После этого программа обрабатывает файл, а затем вызовом *fcntl* снимает с него блокировку.

7.3. API каталогов

Файлы каталогов в UNIX- и POSIX-системах используются для того, чтобы помочь пользователям организовать свои файлы в некую структуру, соответствующую их назначению (например, все исходные тексты программы на C++ можно поместить в каталог */usr/<имя_программы>/C*). Каталоги используются операционной системой также для преобразования путевых имен файлов в номера их индексных дескрипторов.

В BSD UNIX и POSIX.1 каталоги создаются с помощью API *mkdir*:

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

int mkdir ( const char* path_name, mode_t mode ):
```


Аргумент *path_name* — это путевое имя файла каталога, который нужно создать. Аргумент *mode* задает права доступа для владельца, группы и прочих пользователей, которые будут назначены этому файлу. Как и в API *open*, значение *mode* модифицируется значением *umask* вызывающего процесса.

В случае успешного выполнения *mkdir* возвращает 0, а в случае неудачи — -1. Среди возможных причин неудачи следует назвать ошибку в аргументе *path_name*, отсутствие у вызывающего процесса права на создание указанного каталога, ошибку в аргументе *mode*.

В UNIX System V.3 для создания файлов каталогов используется API *mknod*. В UNIX System V.4 поддерживаются оба интерфейса, *mkdir* и *mknod*. Различие между ними состоит в том, что каталог, созданный функцией *mknod*, не содержит ссылок на текущий и родительский каталоги, поэтому его можно использовать только после того, как пользователь создаст эти ссылки явно. Одновременно с созданием каталога функцией *mkdir* появляются и ссылки на текущий и родительский каталоги, поэтому каталог можно использовать сразу же. Вообще-то говоря, *mknod* не следует применять для создания каталогов. А в системах, где не поддерживается API *mkdir*, каталоги можно создавать с помощью API *system*:

```
char   syscmd[256];
sprintf (syscmd, "mkdir %s",<имя_каталога>);
if (system (syscmd) == -1) perror ("mkdir");
```

Идентификатор владельца нового каталога устанавливается равным эффективному идентификатору процесса, его создавшего, а идентификатор группы нового каталога — равным эффективному идентификатору группы вызывающего процесса или идентификатору группы родительского процесса, владеющего новым каталогом (так же, как для обычных файлов).

Каталог — это файл, содержащий записи, в каждой из которых хранится имя и номер индексного дескриптора файла, расположенного в этом каталоге. Структура записей каталога не во всех системах одинакова. Например, в UNIX System V записи каталога имеют фиксированную длину, а в BSD UNIX — переменную. Чтобы процесс мог просматривать каталоги независимо от файловой системы, запись каталога определена как *struct dirent* в заголовке <dirent.h> (в UNIX System V и POSIX.1) и как *struct direct* в заголовке <sys/dir.h> (в BSD UNIX 4.2 и 4.3). Типы данных *struct dirent* и *struct direct* имеют одно общее поле, *d_name*, представляющее собой массив символов, который содержит имя файла, находящегося в каталоге. Для ускоренного просмотра каталогов созданы переносимые функции, определенные в заголовках <dirent.h> и <sys/dir.h>.

```
#include <sys/types.h>
#if defined (BSD) && !_POSIX_SOURCE
    #include <sys/dir.h>
    typedef struct direct Dirent;
#else
    #include <dirent.h>
    typedef stuct dirent Dirent;
#endif

DIR* opendir (const char* path_name);
Dirent* readdir (DIR* dir_fdesc);
int closedir (DIR* dir_fdesc);
void rewinddir (DIR* dir_fdesc);
```

Использование этих функций описано ниже.

Функция	Назначение
opendir	Открывает файл каталога только для чтения. Возвращает указатель на структуру DIR* для последующих обращений к файлу
readdir	Читает запись из файла каталога, обозначенного аргументом *dir_fdesc*, и возвращает содержащуюся в ней информацию
closedir	Закрывает файл, обозначенный аргументом *dir_fdesc*
rewinddir	Устанавливает указатель чтения в начало файла каталога, обозначенного аргументом *dir_fdesc*. Следующий вызов *readdir* прочитает первую запись из этого файла

Функция *opendir* аналогична API *open*. В качестве аргумента она принимает путевое имя файла каталога и открывает этот файл только для чтения. Данная функция возвращает указатель на структуру DIR*, которая используется приблизительно так же, как указатель на структуру FILE*, возвращаемый функцией *fopen*. Структура данных DIR определена в заголовке <dirent.h> или <sys/dir.h>.

Функция *readdir* читает следующую (относительно текущей позиции указателя) запись из файла каталога, обозначенного аргументом *dir_fdesc*. Значение *dir_fdesc* — это значение DIR*, возвращенное вызовом *opendir*. Данная функция возвращает адрес записи типа *struct dirent* или *struct direct*, в которой хранится имя файла. Когда *readdir* вызывается после API *opendir* или *rewinddir*, она возвращает указатель на первую запись данных из файла, при следующем вызове — на вторую запись и т.д. Просмотрев все записи в файле каталога, *readdir* возвращает нулевое значение, показывая тем самым, что достигнут конец файла. Отметим, что тип данных *Dirent* определяется в вышеприведенном прототипе либо как *struct dirent* (для POSIX и UNIX System V), либо как *struct direct* (для BSD UNIX в POSIX-несовместимом режиме). Благодаря этому любое приложение, вызывающее *readdir*, может трактовать возвращаемое значение как *Dirent* независимо от системы, в которой оно работает.

Функция *closedir* аналогична API *close*. Она закрывает соединение между указателем *dir_fdesc* и файлом каталога.

Функция *rewinddir* сбрасывает указатель чтения, связанный с файлом *dir_fdesc*, так, чтобы при следующем вызове *readdir* эта функция могла просматривать данный каталог (обозначенный аргументом *dir_fdesc*) с начала.

В UNIX-системах (System V и BSD UNIX) определены дополнительные функции для произвольной выборки записей файлов каталогов. Стандартом POSIX.1 эти функции не поддерживаются.

Функция	Назначение
telldir	Возвращает указатель на текущую позицию в файле, заданном аргументом *dir_fdesc*
seekdir	Изменяет указатель текущей позиции в файле, заданном аргументом *dir_fdesc*, на указанный адрес

Удаление файлов каталогов производится с помощью API *rmdir*. При наличии статуса привилегированного пользователя можно удалять каталоги с помощью API *unlink*. Эти API требуют, чтобы удаляемые каталоги были пустыми, т.е. не содержали никаких файлов, кроме ссылок на текущий и родительский каталоги. Прототип функции *rmdir* выглядит следующим образом:

```
#include <unistd.h>

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

Ниже приведена программа *list_dir.C*, которая иллюстрирует использование API *mkdir*, *opendir*, *readdir*, *closedir* и *rmdir*

```
#include <iostream.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#if defined (BSD) && !_POSIX_SOURCE
    #include <sys/dir.h>
    typedef struct direct Dirent;
#else
    #include <dirent.h>
    typedef struct dirent Dirent;
#endif

int main (int argc, char* argv[])
{
    Dirent*    dp;
```

```
  DIR*   dir_fdesc;
  while (--argc > 0) { /* выполнить следующее для каждого файла */
    if (!(dir_fdesc=opendir(*++argv)))
    {
      if (mkdir(*argv, S_IRWXU|S_IRWXG|S_IRWXO)==-1)
        perror("opendir");
      continue;
    }
    /* просмотреть каждый файл каталога дважды */
    int i = 0;
    for ( ; i < 2; i++)
    {
      int cnt = 0;
      for (; dp=readdir(dir_fdesc); )
      {
      cout << dp->d_name << endl;
       if (strcmp(dp->d_name,".") && strcmp(dp->d_name,".."))
         cnt++;              /* подсчитать количество файлов
                                в каталоге */
      }
      if (!cnt) { rmdir(*argv); break; }/* пустой каталог */
      rewinddir(dir_fdesc);  /* сбросить указатель для второго цикла */
    }
    closedir(dir_fdesc);
  }                          /* для каждого файла */
  return 0;
}  /* main */
```

В качестве аргументов эта программа принимает один или несколько путевых имен файлов каталогов. Для каждого аргумента программа выполняет указанные ниже операции. Вначале она открывает его вызовом *opendir* и присваивает значение указателя позиции в файле переменной *dir_fdesc*. Если вызов *opendir* неудачен, программа считает, что данного каталога не существует, и пробует создать его с помощью API *mkdir*. Если *opendir* выполняется успешно, программа просматривает файл каталога с помощью API *readdir* и определяет число файлов в нем (без учета файлов "." и ".."). Затем, если каталог пуст, программа удаляет его при помощи API *rmdir*. Если в каталоге есть файлы, то программа с помощью *rewinddir* сбрасывает указатель файла, связанный с *dir_fdesc*, а затем просматривает каталог вторично и выводит имена всех находящихся в нем файлов на стандартный вывод. По завершении второго цикла просмотра каталога программа закрывает *dir_fdesc* с помощью API *closedir*.

7.4. API файлов устройств

Файлы устройств используются для сопряжения физических устройств (например, консоли, модема, дисковода) с прикладными программами. Когда процесс читает файл устройства или записывает в него данные, ядро на основании старшего и младшего номеров устройства выбирает драйвер

для выполнения фактической пересылки данных. Файлы устройств бывают байт-ориентированными и блок-ориентированными.

Поддержка файлов устройств зависит от реализации ОС. В стандарте POSIX.1 порядок создания файлов устройств не оговаривается. В UNIX-системах для создания файлов устройств применяется API *mknod*:

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

int mknod (const char* path_name, mode_t mode, int device_id);
```

Аргумент *path_name* — это путевое имя файла устройства, который нужно создать. Аргумент *mode* задает права доступа к создаваемому файлу для владельца, группы и прочих пользователей, а также устанавливает флаги S_IFCHR или S_IFBLK. Флаги служат для обозначения типа файла устройства (байт-ориентированный или блок-ориентированный). Права доступа модифицируются значением *umask* вызывающего процесса. Наконец, аргумент *device_id* содержит старший и младший номера устройств и в большинстве UNIX-систем строится следующим образом: младший байт *device_id* устанавливается равным младшему номеру устройства, а следующий байт — равным старшему номеру устройства. Например, чтобы создать блок-ориентированный файл устройства *SCSI5* со старшим номером 15, младшим номером 3 и правами доступа на чтение, запись и выполнение для всех пользователей, нужно применить следующий системный вызов *mknod*:

```
mknod("SCSI5", S_IFBLK|S_IRWXU|S_IRWXG|S_IRWXO, (15<<8)|3);
```

В UNIX System V.4 старший и младший номера устройства расширены соответственно до 14 и 18 битов. Старший и младший номера устройства используются следующим образом. Когда процесс читает данные из файла устройства или записывает данные в этот файл, с помощью старшего номера устройства файла производится поиск и вызов драйвера устройства, который выполняет фактический обмен данными с физическим устройством. Младший номер устройства — это аргумент, передаваемый в драйвер устройства при его вызове. Он необходим потому, что драйвер устройства может применяться для устройств различных типов, и младший номер задает параметры (например, размер буфера), которые должны использоваться для организации доступа к конкретному устройству.

API *mknod* должен вызываться процессом с правами привилегированного пользователя. Идентификатор владельца и идентификатор группы для файла устройства назначаются так же, как для обычного файла. Атрибут "размер файла" для файла устройства смысла не имеет.

В случае успешного выполнения функция *mknod* возвращает 0, а в случае неудачи возвращает -1. Среди возможных причин неудачи можно выделить ошибку в путевом имени, отсутствие у процесса права на создание файла устройства, ошибку в аргументе *mode*.

Когда файл устройства будет создан, любой процесс сможет установить с ним соединение, воспользовавшись API *open*. После этого процесс сможет с помощью API *read*, *write*, *stat* и *close* выполнять с файлом необходимые операции. API *lseek* применим только к блок-ориентированным файлам устройств. Удалить файл устройства можно, воспользовавшись API *unlink*.

Когда процесс вызывает функцию *open* для установления соединения с файлом устройства, он может указать флаги O_NONBLOCK и O_NOCTTY, определенные в стандарте POSIX.1. Используются они следующим образом.

Если вызывающий процесс в ОС UNIX не имеет управляющего терминала и открывает байт-ориентированный файл устройства, ядро назначает этот файл устройства управляющим терминалом данного процесса. Если же в вызове *open* установлен флаг O_NOCTTY, это назначение подавляется.

Флаг O_NONBLOCK указывает на то, что вызов *open* и все последующие вызовы *read* и *write* для файла устройства не должны блокировать процесс.

Применять API *mknod* для создания файлов устройств могут только пользователи с особыми правами (например, пользователь *root* в UNIX). Все остальные могут читать и записывать файлы устройств так, как будто это обычные файлы, но с учетом установленных для них прав доступа.

Ниже приведена программа *test_mknod.C*, которая иллюстрирует использование API *mknod*, *open*, *read*, *write* и *close* с байт-ориентированным файлом устройства:

```
#include <iostream.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>

int main( int argc, char* argv[])
{
    if (argc !=4)
    {
      cout << "usage: " << argv[0] << " <file> <major no> <minor no>\n";
      return 0;
    }
    int major = atoi(argv[2]), minor = atoi(argv[3]);
    (void)mknod(argv[1],S_IFCHR|S_IRWXU|S_IRWXG|S_IRWXO,
        (major <8) | minor);
    int rc=1, fd = open(argv[1],O_RDWR|O_NONBLOCK|O_NOCTTY);
    char buf[256];

    while (rc && fd!=-1)
      if ((rc=read(fd,buf,sizeof(buf))) < 0)
          perror("read");
      else if (rc)  cout << buf << endl;
    close(fd);
```



```
    return 0;
}  /* main */
```

Программа принимает три аргумента: имя файла устройства, старший номер устройства и младший номер устройства. Используя названные аргументы, программа создает с помощью вызова *mknod* байт-ориентированный файл устройства. Затем она открывает этот файл для чтения и записи и устанавливает флаги O_NONBLOCK и O_NOCTTY. После этого программа читает из файла данные и направляет их на стандартный вывод. Встретив признак конца файла, программа закрывает дескриптор, связанный с файлом устройства, и завершает выполнение.

Пользователям следует помнить, что файлы устройств обрабатываются почти так же, как обычные файлы. Различия заключаются лишь в способе создания файлов устройств и в невозможности применения к байт-ориентированным файлам устройств вызова *lseek*.

7.5. API FIFO-файлов

FIFO-файлы называют также *именованными каналами*. Это особые файлы *канальных устройств*, используемые для межпроцессного взаимодействия. В частности, процесс может подключиться к FIFO-файлу для чтения, записи и чтения-записи данных. Данные, записываемые в FIFO-файл, хранятся в буфере фиксированного размера (PIPE_BUF, определяется в заголовке <limits.h>) и выбираются по алгоритму "первым пришел — первым вышел".

Для создания FIFO-файлов в BSD UNIX и POSIX.1 определен API *mkfifo*:

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

int mkfifo ( const char* path_name, mode_t mode );
```

Аргумент *path_name* — это путевое имя создаваемого FIFO-файла. Аргумент *mode* задает права доступа к нему для владельца, группы и прочих пользователей, а также устанавливает флаг S_IFIFO, который показывает, что это FIFO-файл. Права доступа модифицируются значением *umask* вызывающего процесса. Идентификаторы владельца и группы для FIFO-файла назначаются так же, как для обычных файлов.

Так, чтобы создать FIFO-файл *FIFO5* с правами доступа на чтение, запись и выполнение для всех пользователей, применяется следующий системный вызов *mkfifo*:

```
mknod("FIFO5", S_IFIFO|S_IRWXU|S_IRWXG|S_IRWXO );
```

В случае успешного выполнения *mkfifo* возвращает 0, в случае неудачи — -1. Среди возможных причин неудачи можно назвать ошибку в путевом

имени, отсутствие у процесса права на создание файла устройства, ошибку в аргументе *mode*.

В UNIX System V.3 для создания FIFO-файлов используется API *mknod*, а UNIX System V.4 поддерживает API *mkfifo*.

После того как FIFO-файл создан, любой процесс имеет право установить с ним соединение, воспользовавшись API *open*. Затем процесс может с помощью интерфейсов *read*, *write*, *stat* и *close* выполнять над файлом необходимые операции. API *lseek* к FIFO-файлам не применим. Удалить FIFO-файл можно посредством API *unlink*.

Если процесс открывает FIFO-файл только для чтения, то ядро блокирует данный процесс до тех пор, пока другой процесс не откроет этот же файл для записи. Если процесс открывает FIFO-файл для записи, то он будет заблокирован до тех пор, пока другой процесс не откроет его для чтения. Такая схема обеспечивает синхронизацию процессов. Кроме того, если процесс попробует записать данные в заполненный FIFO-файл, этот процесс будет заблокирован до тех пор, пока другой процесс не прочитает из файла данные и не освободит таким образом место для новых данных. Если процесс попытается прочитать данные из пустого FIFO-файла, он будет заблокирован до тех пор, пока другой процесс на запишет в этот файл какие-нибудь данные.

Если процессу необходимо избежать блокировки, он может указать в вызове *open* для FIFO-файла флаг O_NONBLOCK. При наличии этого флага API *open* не блокирует процесс, даже если к другому концу канала не подсоединен ни один процесс. Более того, если затем процесс вызовет для такого FIFO-файла API *read* или *write*, а данные для пересылки готовы не будут, эти функции немедленно возвратят значение -1 и установят для кода ошибки значение EAGAIN. В таком случае процесс сможет продолжить выполнение других операций и попробовать вызвать упомянутые функции позже. В UNIX System V определен флаг O_NDELAY, который похож на флаг O_NONBLOCK. Различие между ними состоит в том, что при наличии флага O_NDELAY функции *read* и *write* возвращают нулевое значение в тех случаях, когда они должны блокировать процесс. Однако при этом бывает трудно определить является ли такое их значение признаком конца файла, или же причина состоит в том, что файл пуст (в последнем случае для него сохраняется возможность записи). UNIX System V.4 поддерживает и флаг O_NDELAY, и флаг O_NONBLOCK.

Следует отметить еще один важный момент, касающийся FIFO-файлов. Если процесс записывает данные в FIFO-файл, с которым не взаимодействует в режиме чтения никакой другой процесс, то ядро посылает в него сигнал SIGPIPE (сигналы рассматриваются в главе 9), с тем чтобы уведомить о недопустимости данной операции. Если процесс попробует прочитать FIFO-файл, с которым ни один процесс не взаимодействует в режиме записи, то он прочитает оставшиеся в файле данные и признак конца файла. Таким образом, при взаимодействии двух процессов через FIFO-файл записывающий процесс после завершения работы должен закрыть свой дескриптор файла, так как читающий процесс должен "увидеть" признак конца файла.

Процесс может открыть FIFO-файл для чтения и записи. В стандарте POSIX.1 не указывается, должно ли ядро при этом блокировать процесс, а в UNIX-системах процесс вызовом *open* не блокируется. Для чтения данных из FIFO-файла и записи данных в него процесс может воспользоваться дескриптором файла, возвращенным из API *open*.

На примере программы *test_fifo.C* рассмотрим, как используются API *mkfifo*, *open*, *read*, *write* и *close* с FIFO-файлом:

```
#include <iostream.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>

int main( int argc, char* argv[])
{
    if (argc !=2 && argc !=3)
    {
      cout << "usage: " << argv[0] << " <file> {<arg>}\n";
      return 0;
    }
    int  fd;
    char buf[256];
    (void)mkfifo(argv[1],S_IFIFO|S_IRWXU|S_IRWXG|S_IRWXO);
    if (argc==2)                /* читающий процесс */
    {
      fd = open(argv[1],O_RDONLY|O_NONBLOCK);
      while (read(fd,buf,sizeof(buf))==-1 && errno==EAGAIN)
         sleep(1);
      while (read(fd,buf,sizeof(buf)) > 0)
         cout << buf << endl;
    } else                      /* записывающий процесс */
    {
      fd = open(argv[1],O_WRONLY);
      write(fd,argv[2],strlen(argv[2]));
    }
    close(fd);
    return 0;
}
```

Эта программа принимает один или два аргумента. Первый аргумент — имя используемого FIFO-файла. Если этот файл не существует, программа создает его, вызывая *mkfifo*, а затем проверяет, сколько задано аргументов. Если аргумент один, программа открывает FIFO-файл только для чтения, читает из него все данные и направляет их на стандартный вывод. Если у процесса два аргумента, программа открывает FIFO-файл для записи и записывает в него значение второго аргумента. Таким образом, путем двукратного выполнения этой программы можно создать два процесса,

взаимодействующих через FIFO-файл. Предполагая, что программа компилируется в исполняемый файл *a.out*, получаем следующие результаты:

```
% a.out FF64                    # создать читающий процесс
% a.out FF64 "Hello world"      # создать записывающий процесс
Hello world                     # выходная информация читающего процесса
```

Создавать FIFO-файлы для межпроцессного взаимодействия можно и другим способом — с помощью API *pipe*:

```
#include <unistd.h>

int pipe ( int fds[2] );
```

С помощью API *pipe* можно получить такой же FIFO-файл, как и воспользовавшись *mkfifo*, однако файл, созданный первым способом, будет нерезидентным: в файловой системе не создается никакого файла, связанного с FIFO-файлом, и ядро уничтожает его как только все процессы закрывают свои дескрипторы файлов, ссылающиеся на этот FIFO. Аргумент *fds* используется следующим образом: *fds[0]* — это дескриптор файла для чтения данных из FIFO-файла, а *fds[1]* — дескриптор файла для записи данных в FIFO-файл. Поскольку FIFO-файл нельзя обозначить путевым именем, область его применения ограничена родственными процессами: FIFO-файл создается родительским процессом, который затем порождает другие процессы; процессы-потомки наследуют от родителя дескрипторы FIFO-файла и могут через этот файл взаимодействовать друг с другом и с родительским процессом. Такое ограничение по применению FIFO-файлов, создаваемых API *pipe*, привело к появлению именованных каналов, которые позволяют общаться через FIFO-файлы "неродственным" процессам.

7.6. API символических ссылок

Символические ссылки определены в BSD UNIX 4.2 и используются в BSD 4.3, System V.3 и V.4. Символические ссылки дают возможность устранить некоторые недостатки, характерные для жестких ссылок, а именно:

- их можно создавать для файлов, находящихся в другой файловой системе;
- позволяют ссылаться на файл каталога;
- всегда указывают последнюю версию файла.

Возможность узнать последнюю версию файла — это главное преимущество символических ссылок по сравнению с жесткими. Пусть, например, пользователь создаст файл */usr/go/test1* и жесткую ссылку на него */usr/joe/hdlnk*:

```
ln  /usr/go/test1  /usr/joe/hdlnk
```


Если пользователь удалит файл */usr/go/test1*, то к нему можно будет обращаться только по имени */usr/joe/hdlnk*. Если затем пользователь создаст новый файл */usr/go/test1*, который будет полностью отличаться от файла */usr/joe/hdlnk*, то имя */usr/joe/hdlnk* все равно будет обозначать старый файл, а имя */usr/go/test1* — новый. Таким образом, путем удаления одной или нескольких ссылок жесткие ссылки можно разрушать.

Символическая ссылка в отличие от жесткой не разрушается. Если пользователь, создав символическую ссылку */usr/joe/symlnk*:

```
ln  -s  /usr/go/test1  /usr/joe/symlink
```

удалит */usr/go/test1*, то имя */usr/joe/symlnk* будет обозначать несуществующий файл и ни одна операция с этой ссылкой (*cat*, *more*, *sort* и т.д.) выполняться не будет. Если же пользователь создаст файл */usr/go/test1* заново, то */usr/joe/symlnk* автоматически укажет на этот новый файл, т.е. ссылка будет восстановлена.

Символические ссылки предложено включить в стандарт POSIX.1. В BSD UNIX для манипулирования символическими ссылками определены следующие API:

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

int symlink ( const char* org_link, const char* sym_link );
int readlink ( const char* sym_link, char* buf, int size );
int lstat ( const char* sym_link, struct stat* statv );
```

Аргументы *org_link* и *sym_link* в вызове *symlink* задают путевое имя исходного файла и путевое имя создаваемой символической ссылки. Например, чтобы создать символическую ссылку */usr/joe/lnk* для файла */usr/go/test1*, следует воспользоваться следующим вызовом:

```
symlink ("/usr/go/test1", "/usr/joe/lnk");
```

Синтаксис вызова такой же, какой применяется для API *link*. В случае успешного выполнения *symlink* возвращает 0, а в случае неудачи возвращает -1. Возможные причины сбоя: неверно заданное путевое имя, наличие файла *sym_link*, отсутствие у вызывающего процесса права на создание нового файла.

Чтобы запросить путевое имя, на которое указывает символическая ссылка, следует использовать API *readlink*. API *open* автоматически преобразует символическую ссылку в реальный файл, который она обозначает, и соединяет вызывающий процесс с этим файлом. API *readlink* принимает следующие аргументы: *sym_link* — путевое имя символической ссылки; *buf* — буфер массива символов, в котором хранится возвращаемое путевое имя, обозначенное символической ссылкой; *size* — максимальная емкость (в байтах)

аргумента *buf*. В случае успешного выполнения *readlink* возвращает количество символов путевого имени, помещенного в аргумент *buf*, а в случае неудачи возвращает -1. Возможные причины неудачи: путевое имя *sym_link* не является символической ссылкой, аргумент *buf* представляет собой неверный адрес, у вызывающего процесса нет прав доступа к символической ссылке.

Следующая функция принимает в качестве аргумента путевое имя символической ссылки и многократно вызывает *readlink* для получения имени файла, на который указывает эта ссылка. Цикл *while* завершается, когда *readlink* возвращает -1, при этом переменная *buf* содержит путевое имя реального файла, которое затем направляется на стандартный вывод:

```
#include <iostream.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>

int resolve_link( const char* sym_link )
{
   char* buf[256], tname[256];
   strcpy(tname, sym_link);
   while ( readlink( tname, buf, sizeof( buf ) ) > 0 )
      strcpy( tname, buf );
   cout << sym_limk << "=>" << buf <<endl;
}
```

Функция *lstat* используется для получения атрибутов файлов символических ссылок. Она необходима по той причине, что функции *stat* и *fstat* сообщают атрибуты только несимволических ссылок. Прототип и возвращаемые значения функции *lstat* — такие же, как и у функции *stat*. Более того, *lstat* можно использовать и с несимволическими ссылками, поскольку она работает как *stat*. UNIX-команда *ls -l* использует *lstat* для отображения информации о файлах всех типов, включая символические ссылки.

Ниже приведена программа *test_symln.C*, которая эмулирует UNIX-команду *ln*. Основное назначение этой программы — создание ссылок на файл. Имена исходного файла и новой ссылки являются аргументами. Если опция *-s* не указана, программа создает жесткую ссылку, в противном случае — символическую:

```
#include <iostream.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>

/* Эмуляция UNIX-команды ln */
int main (int argc, char* argv[])
{
   char* buf[256], tname[256];
   if ((argc< 3 && argc > 4) || (argc==4 && strcmp(argv[1],"-s"))) {
      cout << "usage: " << argv[0] << " [-s] <orig_file> <new_link>\n";
```



```
      return 1;
   }
   if (argc==4)
      return symlink( argv[2], argv[3]); /* создает символическую
                                            ссылку */
   else
      return link(argv[1], argv[2]);     /* создает жесткую ссылку */
   return 0;
}
```

7.7. Общий класс для файлов

Класс *fstream* в языке C++ служит для определения объектов, которые служат для представления файлов. В частности, класс *fstream* содержит функции-члены *open*, *close*, *read*, *write*, *tellg* и *seekg*, которые построены на API *open*, *read*, *write* и *lseek*. Таким образом, любая прикладная программа может определить объекты класса *fstream*, предназначенные для чтения и записи файлов.

Следует отметить, однако, что в классе *fstream* нет никаких средств для выполнения функций *stat*, *chmod*, *chown*, *utime* и *link* с его объектами. Кроме того, здесь могут создаваться только обычные файлы. Таким образом, *fstream* не обеспечивает выполнения полного набора характерных для POSIX- и UNIX-систем функций, предназначенных для работы с файловыми объектами.

Чтобы устранить этот недостаток, ниже определяется новый класс, *filebase*, который обладает свойствами класса *fstream* и содержит дополнительные функции, позволяющие пользователям получать атрибуты файловых объектов, изменять их и создавать жесткие ссылки:

```
#ifndef FILEBASE_H      /* filebase.h header */
#define FILEBASE_H

#include <iostream.h>
#include <fstream.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <utime.h>
#include <fcntl.h>
#include <string.h>
typedef enum  { REG_FILE='r', DIR_FILE='d', CHAR_FILE='c',
                BLK_FILE='b', PIPE_FILE='p', SYM_FILE='s',
                UNKNOWN_FILE='?'}  FILE_TYPE_ENUM;

/* Базовый класс, инкапсулирующий свойства файловых объектов POSIX и UNIX */
class filebase:   public fstream
{
   protected:
     char* filename;
       friend ostream& operator<<(ostream& os, filebase& fobj)
     {
```

```
      /* Вывести атрибуты файла в формате UNIX-команды ls -l */
       return os;
    };
  public:
    filebase()     {;};
     filebase(const char* fn, int flags, int prot=filebuf::openprot)
       : fstream(fn,flags,prot)
    {
       filename = new char[strlen(fn)+1];
       strcpy(filename,fn);
    };
    virtual ~filebase()  { delete filename; };
    int  fileno() { return rdbuf()->fd(); );
    virtual int create( const char* fn, mode_t mode)
       { return ::creat(fn, mode); };
    int chmod(mode_t mode)
       { return ::chmod (filename, mode); );
    int chown( uid_t uid, gid_t gid )
       { return ::chown(filename, uid, gid); );
    int utime(const struct utimbuf *timbuf_Ptr)
       { return ::utime(filename,timbuf_Ptr); };
    int link( const char* new_link)
       { return ::link(filename,new_link); );
     virtual int remove( )
       { return ::unlink(filename); );
     // Запросить тип файла для объекта класса filebase
     FILE_TYPE_ENUM file_type()
     {
        struct stat statv;
        if (stat(filename,&statv)==0)
          switch (statv.st_mode & S_IFMT)
          {
              case S_IFREG: return REG_FILE;   // обычный файл
              case S_IFDIR: return DIR_FILE;   // каталог
              case S_IFCHR: return CHAR_FILE;  // байт-ориентированный
                                               // файл устройства
              case S_IFIFO: return PIPE_FILE;  // блок-ориентированный
                                               // файл устройства
              case S_IFLNK: return SYM_FILE;   // символическая ссылка
          }
       return UNKNOWN_FILE;
     };
};
#endif      /* filebase.h */
```

Конструктор *filebase* передает свои аргументы конструктору своего надкласса *fstream* для соединения (с заданным режимом доступа) объекта с файлом, указанным в аргументе *fn*. Затем конструктор автоматически выделяет динамический буфер для хранения путевого имени *fn* в закрытой переменной *filename*. Переменная *filename* используется в других функциях-членах, таких как *chmod*, *link*, *remove* и т.д.

Деструктор *filebase* освобождает динамический буфер *filename*, а затем с помощью деструктора *fstream* отсоединяет объект от файла, заданного аргументом *filename*.

Функция-член *fileno* возвращает дескриптор файла, которым управляет объект класса *filebase*.

Функция-член *chmod* помогает пользователям изменить права доступа к файлу, соединенному с объектом класса *filebase*. Аргумент *mode* — такой же, как у функции *chmod* стандарта POSIX.1, а переменная *filename* определяет, права доступа к какому файлу нужно изменить.

Функция-член *chown* помогает пользователям изменить идентификатор владельца и идентификатор группы для файла, соединенного с объектом класса *filebase*. Аргументы *uid* и *gid* — те же, что у функции *chown* стандарта POSIX.1; переменная *filename* показывает, принадлежность какого файла нужно изменить.

Функция-член *utime* помогает пользователям изменить дату и время доступа и модификации файла, соединенного с объектом класса *filebase*. Аргумент *timbuf_Ptr* — такой же, как у функции *utime* стандарта POSIX.1, а переменная *filename* определяет, у какого файла нужно изменить эти атрибуты.

Функция-член *link* позволяет пользователям создать жесткую ссылку на файл, соединенный с объектом класса *filebase*. Аргумент *new_link* содержит имя новой ссылки. Первоначальная ссылка на файл, соединенный с объектом, берется из переменной *filename* данного объекта. Для создания новой жесткой ссылки эта функция вызывает функцию *link* стандарта POSIX.1.

Функция-член *create* создает файл с заданным именем. Для этого ей необходимо вызвать функцию API *creat*. Функция-член *create* применима только к обычным файлам.

Функция-член *remove* удаляет ссылку, заданную переменной *filename* объекта. При успешном завершении такого вызова ни один процесс не сможет обращаться к данному файлу по этому путевому имени.

Перегруженная операция "<<" в *ostream* используется для вывода атрибутов файла, соединенного с объектом класса *filebase*. Эта функция может вызывать функцию *long_list* (см. раздел 7.1.10) для определения свойств файла, имя которого задано закрытой переменной *filename*.

Функция *file_type* определяет тип объекта класса *filebase*. Она вызывает API *stat* для заданного *filename* и определяет тип соответствующего файлового объекта. Эта функция возвращает данные типа *FILE_TYPE_ENUM* (номер, соответствующий типу файла). Если вызов *stat* неудачен или если тип файла объекта не определен, эта функция возвращает значение UNKNOWN_FILE. В противном случае она возвращает значение REG_FILE, DIR_FILE или какое-либо другое.

Ниже приведена программа *test_filebase.C*, которая иллюстрирует использование класса *filebase*. Здесь определяется объект класса *filebase* с именем *rfile*, который необходимо связать для чтения с файлом */usr/text/unix.doc*. Затем программа выдает атрибуты файла на стандартный вывод и заменяет

идентификатор пользователя и идентификатор группы файла соответственно на 15 и 30. Далее программа изменяет время доступа и модификации файла на текущее и создает жесткую ссылку */home/jon/hdlnk*. Наконец, удаляется первоначальная ссылка */usr/text/unix.doc*.

```
#include "filebase.h"
int main()
{                                        // определить объект
    filebase rfile("/usr/text/unix.doc",ios::in);

    cout << rfile << endl;               // вывести атрибуты файла
    rfile.chown(15, 30);                 // изменить UID и GID
    rfile.utime(0);                      // изменить время доступа
    rfile.link("/home/jon/hdlnk");       // создать жесткую ссылку
    rfile.remove();                      // удалить старую ссылку
    return 0;
}
```

В классе *filebase* определяются родовые функции для всех типов файлов, допустимых в POSIX и UNIX. Следует отметить, однако, что здесь нет никаких средств для создания файлов, отличных от обычных, и не осуществляется поддержка операций, зависящих от типа файла (например, не поддерживается блокировка файлов). Чтобы устранить эти недостатки, были созданы новые подклассы класса *filebase*, которые обеспечивают полную инкапсуляцию данных для файлов UNIX и POSIX различных типов. Эти подклассы описаны в следующих разделах.

7.8. Класс regfile для обычных файлов

Класс *filebase* инкапсулирует большинство свойств и функций, необходимых для представления обычных файловых объектов в POSIX- и UNIX-системах, за исключением блокировки файлов. Класс *regfile* определяется как подкласс класса *filebase*, но включает функции блокировки файлов. Таким образом, объекты класса *regfile* могут выполнять все операции над обычными файлами, допустимые в POSIX и UNIX.

Класс *regfile* определяется следующим образом:

```
#ifndef REGFILE_H
#define REGFILE_H
#include "filebase.h"
/* Класс, инкапсулирующий свойства обычных файловых объектов POSIX
   и UNIX */
class regfile:   public filebase
{
   public:
     regfile( const char*fnm, int mode, int prot) :
 filebase(fnm,mode,prot)
                    {};
     ~regfile()     {};
```



```
     int lock( int lck_type, off_t len, int cmd=F_SETLK)
            (   struct flock flck;
                if ((lck_type&ios::in) == ios::in)
                   flck.l_type = F_RDLCK;
                else if ((lck_type & ios::out)==ios::out)
                   flck.l_type = F_WRLCK;
                else return -1;
                flck.l_whence = SEEK_CUR;
                flck.l_start = (off_t)0;
                flck.l_len = len;
                return fcntl(fileno(),cmd,&flck);
            };
     int lockw( int lck_type, off_t len)
            (   return lock(lck_type, len, F_SETLKW);  };
     int unlock( off_t len)
            (   struct flock flck;
                flck.l_type = F_UNLCK;
                flck.l_whence = SEEK_CUR;
                flck.l_start = (off_t)0;
                flck.l_len = len;
                return fcntl(fileno(),F_SETLK,&flck);
            };
      int getlock( int lck_type, off_t len, struct flock& flck)
            (   if ((lck_type&ios::in) == ios::in)
                  flck.l_type = F_RDLCK;
                else if ((lck_type & ios::out)==ios::out)
                  flck.l_type = F_WRLCK;
                else return -1;
                flck.l_whence = SEEK_CUR;
                flck.l_start = (off_t)0;
                flck.l_len = len;
                return fcntl(fileno(),F_GETLK,&flck);
             };
};
#endif        /* regfile.h */
```

Функция *regfile::lock* устанавливает блокировку чтения (*lck_type = ios::in*) или блокировку записи (*lck_type = ios::out*) на область файла, связанного с объектом класса *regfile*. Начальным адресом блокируемой области является текущий указатель позиции в файле. Его можно установить функцией *fstream::seekg*. Размер блокируемой области в байтах задается аргументом *len*. Функция *regfile::lock* по умолчанию не блокирует вызывающий процесс (т.е. является неблокирующей). Если задать аргумент *cmd* как F_SETLKW или вызвать эту функцию из функции *regfile::lockw*, функция будет блокирующей. Возвращаемое значение функции *regfile::lock* то же самое, что и для функции *fcntl* в операции блокировки файла.

Функция *regfile::lockw* — это оболочка функции *regfile::lock*. Она работает в блокирующем режиме. Возвращаемое ею значение совпадает с возвращаемым значением функции *regfile::lock*.

Функция *regfile::unlock* освобождает область файла, связанного с объектом класса *regfile*. Начальным адресом освобождаемой области является текущий указатель позиции в файле. Его можно установить функцией *fstream::seekg*. Размер заблокированной области (в байтах) задается аргументом *len*. Функция *regfile::unlock* по умолчанию является неблокирующей и возвращает то же значение, что и *fcntl* в режиме снятия блокировки с файлов.

Функция *regfile::getlock* запрашивает информацию о режиме блокировки указанной в вызове этой функции области файла, связанного с объектом класса *regfile*. Аргумент *lck_type* показывает, о какой блокировке необходима информация, — о блокировке чтения (*lck_type* = *ios::in*) или о блокировке записи (*lck_type* = *ios::out*). Начальным адресом области является текущий указатель позиции в файле. Его можно установить с помощью функции *fstream::seekg*. Размер области (в байтах) задается аргументом *len*. Функция *regfile::getlock* является неблокирующей и возвращает то же значение, что и функция *fcntl*. При успешном выполнении вызова функции *regfile::getlock* аргумент *flck* будет содержать информацию о блокировках для данной области файла.

Класс *regfile* обеспечивает полную инкапсуляцию всех функций, предусмотренных в UNIX и POSIX для обычных файлов. Он также упрощает API блокировки и освобождения файлов, чтобы для работы с функциями *regfile::lock*, *regfile::unlock* и *regfile::getlock* пользователям достаточно было знать только интерфейс класса *fstream*. Ниже приведена программа *test_regfile.C*, которая иллюстрирует использование объекта *regfile* для создания временного файла *foo*, блокирует весь этот файл по записи, инициализирует его содержимое данными из файла */etc/passwd*, освобождает первые 10 байтов файла и, наконец, удаляет его:

```
#include  "regfile.h"
int main()
{
    ifstream  ifs {"/etc/passwd");
    char     buf[256];
    regfile  rfile{"foo",ios::out | ios::in, 0777);
                                  // определить объект класса regfile
    rfile.lock{ios::out,0);       // заблокировать файл по записи
    while {ifs.getline{buf,256)) rfile << buf << endl;
    rfile.seekg(0,ios::beg);      // установить указатель позиции в
                                  // начало файла
    rfile.unlock{10);             // разблокировать первые 10 байтов
                                  // файла
    rfile.remove();               // удалить файл
    return 0;
}
```


7.9. Класс dirfile для каталогов

Класс *dirfile* определен с целью инкапсуляции всех функций, предусмотренных для файлов каталогов в UNIX и POSIX. В частности, в классе *dirfile* определены функции *create*, *open*, *read*, *tellg*, *seekg*, *close* и *remove*, которые используют API каталогов, применяемые в UNIX и POSIX.

Класс *dirfile* определяется следующим образом:

```
#ifndef DIRFILE_H
#define DIRFILE_H
#include "filebase.h"
#include <dirent.h>
#include <string.h>
/* класс, инкапсулирующий свойства обычных файловых объектов POSIX и UNIX */
class dirfile
{
      DIR *dir_Ptr;
      char    *filename;
  public:
      dirfile(const char* fn)   {  dir_Ptr = opendir{fn);
                  filename = new char[strlen{fn)+1];
                  strcpy{filename,fn);
                  );
      ~dirfile{)        (   if (dir_Ptr) close();
                        delete filename;
                  };
      int close()       (
                  return {dir_Ptr) ? closedir{dir_Ptr) : -1;
                  ];
      int create( const char* fn, mode_t prot)
               {  return mkdir{fn,prot);  };
      int open{const char* fn) {   dir_Ptr=opendir{fn);
                        return dir_Ptr ? 0 : -1;
                        };
      int read{char* buf, int size)
                  {
                  struct dirent *dp = readdir{dir_Ptr);
                  if {dp)
                     strncpy{buf,dp->d_name,size);
                  return {dp)? strlen(dp->d_name) : 0;
                  };
     off_t tellg()      {  return telldir(dir_Ptr); };
     void seekg( off_t ofs )    (  seekdir(dir_Ptr, ofs); ];
     int remove()       {   return rmdir(filename); ];
};
#endif      /* dirfile.h */
```

Для создания файлов каталогов в классе *dirfile* используется API *mkdir*. Кроме того, для открытия и чтения каталогов применяются API *opendir* и *readdir*. Объект класса *dirfile* может трактоваться пользователем почти так же, как обычный файловый объект. Единственное их различие состоит в том,

что в объекте класса *dirfile* ничего не предусмотрено для операций записи. Функции *dirfile::tellg* и *dirfile::seekg*, используя UNIX-интерфейсы *telldir* и *seekdir*, обеспечивают произвольный доступ к любой записи каталога. Наконец, функция *dirfile::close* с помощью *closedir* закрывает файл каталога, а функция *dirfile::remove*, применив API *rmdir*, удаляет этот файл из файловой системы.

Ниже приведена программа *test_dirfile.C*, которая открывает каталог */etc* и выводит на экран перечень всех находящихся в нем файлов:

```
#include "dirfile.h"
int main()
{
  dirfile   edir("/etc");     // создать объект класса dirfile для /etc
  char  buf[256];
  while (edir.read(buf,256)) // показать файлы, имеющиеся в каталоге /etc
    cout << buf << endl;
  edir.close();               // закрыть каталог
  return 0;
]
```

7.10. Класс pipefile для FIFO-файлов

Файловый объект типа FIFO отличается от объекта класса *filebase* тем, что создается в процессе выполнения программы и функции *tellg* и *seekg* для него не приемлемы. Класс *pipefile*, определение которого приведено ниже, инкапсулирует все свойства FIFO-файлов

```
#ifndef PIPEFILE_H
#define PIPEFILE_H
#include "filebase.h"

/* класс, инкапсулирующий свойства обычных FIFO-файловых объектов
   POSIX и UNIX */
class pipefile   : public filebase
{
   public:
      pipefile(const char* fn, int flags, int prot) :
               filebase(fn,flags,prot) {];
      int create( const char* fn, mode_t prot)
               { return mkfifo(fn,prot); ];
      streampos tellg() {  return (streampos)-1; };
};
#endif     /* pipefile.h */
```

Ниже приведена программа *test_pipefile.C*, которая создает FIFO-файл с именем FIFO, открывает его для чтения (если *argc* = 1) или для записи (если *argc* > 1). Затем программа читает или записывает данные с использованием FIFO-файла и закрывает его:


```
#include     "pipefile.h"
int main( int argc, char* argv[] )
{
pipefile  nfifo("FIFO", argc==1 ? ios::in : ios::out, 0755);
if (argc > 1)              // записывающий процесс
{
   cout << "writer process write: " << argv[1] << endl;
     nfifo.write(argv[1],strlen(argv[1])+1);// write data to FIFO
}  else                    // читающий процесс
{
    char buf[256];
    nfifo.read(buf,256);   // прочитать данные из FIFO-файла
    cout << "read from FIFO: " << buf << endl;
]
nfifo.close();             // закрыть FIFO-файл
return 0;
}
```

Путем повторного выполнения этой программы можно создать два процесса, которые будут взаимодействовать через FIFO-файл. Сначала программа выполняется без аргумента командной строки и создает читающий процесс, затем запускается с аргументом командной строки, который создает записывающий процесс. После того как оба процесса будут созданы, записывающий процесс запишет свой аргумент командной строки в FIFO-файл, а читающий процесс прочитает этот аргумент из файла и выдаст его на стандартный вывод. Ниже приведены результаты выполнения этой программы:

```
% CC -o test_pipefile test_pipefile.C
% test_pipefile &              # создать читающий процесс
% test_pipefile "hello"        # создать записывающий процесс
writer process write: hello    # выходная информация записывающего
                                 процесса
read from FIFO: hello          # выходная информация читающего процесса
```

7.11. Класс devfile для файлов устройств

Объект типа "файл устройства" обладает всеми свойствами обычного файлового объекта, за исключением метода создания. Кроме того, применение функций *tellg*, *seekg*, *lock*, *lockw*, *unlock* и *getlock* для байт-ориентированных файлов устройств не допустимо. Определенный ниже класс *devfile* инкапсулирует все свойства UNIX-файлов устройств:

```
#ifndef DEVFILE_H
#define DEVFILE_H
#include "regfile.h"

/* класс, инкапсулирующий свойства файлов устройств POSIX- и
   UNIX-систем */
class devfile  : public regfile
{
```

```
public:
   devfile(const char* fn, int flags, int prot) :
     regfile(fn,flags,prot)  {};
   int create(const char* fn, mode_t prot, int major_no,
        int minor_no, char type='c')
  {  if (type=='c')
          return mknod(fn, S_IFCHR | prot,
             (major_no << 8)  | minor_no);
      else return mknod(fn, S_IFBLK | prot,
             (major_no < 8)  | minor_no);
  };
  streampos  tellg()      {  if  (file_type()==CHAR_FILE)
                   return  (streampos)-1;
                else  return  fstream::tellg();
                };
  istream& seekg( streampos ofs, seek_dir d)
          {    if  (file_type()!=CHAR_FILE)
                 fstream::seekg(ofs,d);
                return *this;
          };
  int lock( int lck_type, off_t len, int cmd=F_SETLK)
          {     if (file_type()!=CHAR_FILE)
                  return
                    regfile::lock(lck_type,len,cmd);
                else return -1;
          };
   int lockw( int lck_type, off_t len)
          {     if  (file_type()!=CHAR_FILE)
                  return
                    regfile::lockw(lck_type,len);
                else return -1;
          };
  int unlock( off_t len)
          {    if  (file_type()!=CHAR_FILE)
                  return  regfile::unlock(len);
               else return -1;
          };
   int getlock( int lck_type, off_t len, struct flock& flck)
          {    if  (file_type()!=CHAR_FILE)
                 return
                  regfile::getlock(lck_type,len,flck);
               else return -1;
               };
};
#endif     /* devfile.h */
```

Функция *devfile::create* создает файл устройства с помощью API *mknod*. Аргумент *type* показывает, какой файл устройства должен быть создан — байт-ориентированный (*type=c*) или блок-ориентированный (*type=b*). Старший и младший номера устройства, права доступа и имя файла устройства задаются аргументами *major_no*, *minor_no*, *prot* и *fn*. Функция *devfile::create* возвращает значение, равное возвращаемому значению API *mknod*.

Созданный файл устройства можно открывать, читать, записывать в него данные, закрывать его подобно обычному файлу. Таким образом, *devfile*

наследует все функции *fstream*, необходимые для информационного доступа к объектам. При этом произвольная выборка данных из байт-ориентированных файлов устройств не допускается. Следовательно, функции *devfile::tellg* и *devfile::seekg* работают только с блок-ориентированными файлами устройств.

Ниже приведена программа *test_devfile.C*, создающая байт-ориентированный файл устройства, который имеет имя */dev/tty*, а также старший и младший номера, соответственно равные 30 и 15. Затем программа открывает этот файл для записи данных и закрывает его.

```
#include "devfile.h"
int main()
{                                  // открыть файл устройства для записи
devfile ndev("/dev/tty", ios::out, 0755);
ndev << "This is a sample output string\n";
                                   // записать данные в файл
ndev.close();                      // закрыть файл устройства
return 0;
}
```

7.12. Класс symfile для символических ссылок

Объект типа символическая ссылка отличается от объекта класса *filebase* способом создания. Кроме того, имеется новая функция-член *ref_path*, которая описывает путевое имя файла, являющегося объектом-символической ссылкой. Определенный ниже класс *symfile* инкапсулирует все свойства символических ссылок UNIX

```
#ifndef SYMFILE_H  /* This is symfile.h header */
#define SYMFILE_H
#include  "filebase.h"
/* A class to encapsulate UNIX symbolic link file objects'
   properties */
class symfile:  public filebase
{
    public:
       symfile()    {};
       symfile(const char* fn, int flags, int prot) :
filebase(fn,flags,prot)    {;};
       void open( int mode ) {  fstream::open( filename, mode ); }
       int setlink( const char* old_link, const char* new_link)
              {   filename = new char[strlen(new_link)+1];
                   strcpy(filename,new_link);
                  return symlink(old_link,new_link);
              };
       const char* ref_path() {  static char xbuf[256];
                    if (readlink(filename,xbuf,256))
                      return xbuf;
                    else return (char*)-1;
                    };
```

```
};
#endif   /* symfile.h */
```

Функция *symfile::create* создает символическую ссылку с помощью API *symlink*. Аргумент *new_link* — это путевое имя новой символической ссылки, которая должна быть создана, а *old_link* — путевое имя исходного файла. Функция *symfile::create* возвращает то же значение, что и функция API *symlink*.

Символическую ссылку можно открывать, читать, записывать в нее данные и закрывать как обычный файл. Таким образом, *symfile* наследует все функции *fstream*, необходимые для информационного доступа к ее объектам. Все операции выполняются с реальным файлом, на который указывает символическая ссылка. Кроме того, определена функция *symfile::ref_path*, с помощью которой пользователи могут запрашивать путевое имя реального файла, на который указывает объект класса *symfile*.

Ниже приведена программа *test_symfile.C*, предназначенная для создания символической ссылки */usr/xyz/sym.lnk*, которая обозначает файл */usr/file/chap10*. Программа открывает файл и читает его содержимое. Затем она выводит на экран путевое имя ссылки и закрывает файл:

```
#include "symfile.h"
int main()
{
char  buf[256];
symfile nsym;
nsym.setlink( "/usr/file/chap10", "/usr/xyz/sym.lnk" );
nsym.open( ios::in );
while (nsym.getline(buf,256))
  cout << buf << endl;                // прочитать /usr/file/chap10
cout << nsym.ref_path() << endl;      // вывести на экран имя файла
                                      // "/usr/file/chap10"
nsym.close();                         // закрыть файл, представляющий
                                      // собой символическую ссылку
return 0;
}
```

7.13. Программа вывода на экран списка файлов

В качестве примера использования класса *filebase* и его подклассов ниже приведена программа *lstdir.C*, которая эмулирует UNIX-команду *ls*, с тем чтобы получить список атрибутов всех файлов, заданных аргументами программы. Если аргумент — это каталог, то программа выдает список атрибутов всех файлов, содержащихся в данном каталоге и его подкаталогах. Если файл является символической ссылкой, программа выдает путевое имя, на которое указывает эта ссылка. Таким образом, рассматриваемая программа работает как UNIX-команда *ls -lR*.

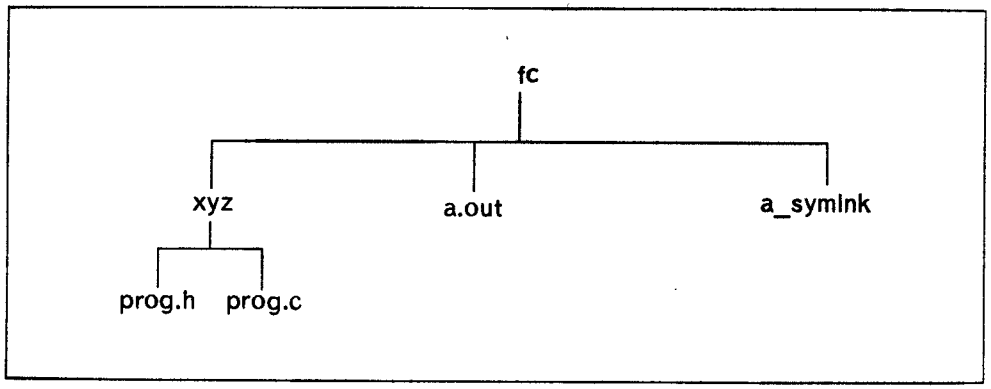

```
#include "filebase.h"
#include "symfile.h"
#include "dirfile.h"

extern void long_list( ostream& ofs, const char* fname );
void show_list( ostream& ofs, char* fname, int deep);

/* программа реализации UNIX-команды ls -lR */
void show_dir( ostream& ofs, const char* fname )
{
   dirfile  dirObj(fname);
   char  buf[256];
   ofs << "\nDirectory: " << fname << ":\n";
   while (dirObj.read(buf,256)) // показать все файлы в каталоге
      show_list(ofs, buf,0);
   dirObj.seekg(0);             // установить указатель позиции
                                // в начало файла
   while (dirObj.read(buf,256)) // искать каталоги
   {
      filebase    fObj(buf,ios::in,0755);
                                // определить объект класса filebase
      if (fObj.file_type()==DIR_FILE)
                                // выдать информацию о подкаталогах
         show_dir(ofs,buf);
       fObj.close();
   }
   dirObj.close();
}

void show_list( ostream& ofs, const char* fname, int deep)
{
   long_list( ofs, fname);
   filebase    fobj(fname,ios::in,0755);
   if (fobj.file_type()==SYM_FILE) // символическая ссылка
   {
      symfile  *symObj = (symfile*)&fobj;
                                   // определить объект класса symfile
      ofs << " -> " << symObj->ref_path() << endl;
                                   // показать путевое имя ссылки
   }
   else if (fobj.file_type() == DIR_FILE && deep)// каталог
       show_dir( ofs, fname );     // показать содержимое каталога
}

int main( int argc, char* argv[])
{
   while (--argc > 0) show_list( cout , *++argv, 1);
   return 0;
}
```

Эта программа похожа на программу, приведенную в разделе 7.1.10, за исключением того, что она выдает символические ссылки и перечисляет

содержимое каталога рекурсивно. Функция *main* обрабатывает каждый аргумент командной строки, вызывая функцию *show_file* для отображения атрибутов каждого файла на стандартном выводе. Аргумент *deep* функции *show_file* указывает (если он имеет ненулевое значение), что если аргумент *fname* — каталог, *show_file* должна вызвать функцию *show_dir* для построения списка содержимого этого каталога. Когда *show_file* вызывается из головного модуля *main*, фактическое значение *deep* устанавливается равным 1.

Функция *show_dir* вызывается для вывода содержимого каталога. Сначала она создает объект класса *dirfile* для связывания его с каталогом, имя которого задано аргументом *fname*. Затем она вызывает функцию *show_file*, перечисляющую атрибуты каждого файла данного каталога. В этом первом проходе *show_file* вызывается с нулевым значением аргумента *deep*, чтобы *show_file* при вызове *show_dir* не выводила содержимое подкаталогов. После завершения первого прохода *show_dir* просматривает данный каталог во второй раз и ищет подкаталоги. Для каждого обнаруженного подкаталога эта функция вызывает себя рекурсивно и выдает список содержимого подкаталога. Именно так ведет себя UNIX-команда *ls -lR*.

Рассмотрим в качестве примера следующую структуру каталогов:



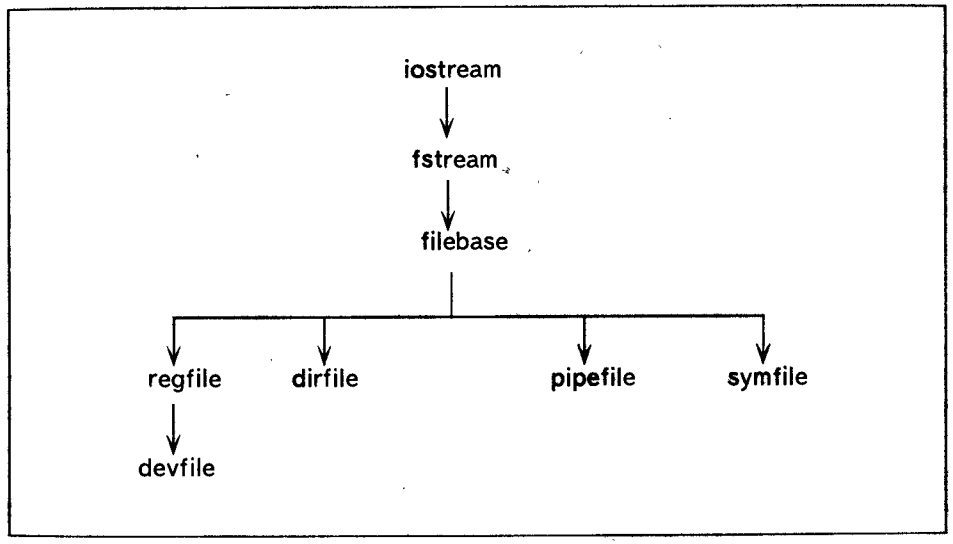
В результате выполнения программы *lstdir* получаем следующее:

```
% CC -o lstdir lstdir.C
% lstdir tc
-r-xr-xr-x     1   util    class    1234 Apr 8, 1997 a.out
lrwxrwxrwx     1   util    class    122  Apr 11, 1997 a_symlink ->
/usr/dsg/unix.lnk
drwxr-x--x     1   util    class    234  Jan 17, 1997 xyz

Directory: xyz
-rw-r--r--     1   util    class    814  Dec 18, 1996 prog.c
-rwxrwxrwx     1   util    class    112  May 21, 1997 prog.h
```


7.14. Заключение

В этой главе описаны файловые API операционной системы UNIX и стандарта POSIX. Эти интерфейсы прикладного программирования используются для создания, открытия, чтения, записи и закрытия файлов всех типов: обычных, каталогов, файлов устройств, FIFO-файлов и символических ссылок. Для инкапсуляции свойств и функций файлов всех типов определен набор классов C++, с помощью которых пользователи могут манипулировать файлами, используя тот же интерфейс, который применяется для класса *iostream*. Все эти классы переносимы на все UNIX- и POSIX-системы. Исключение составляет класс символических ссылок (*symfile*), который в стандарте POSIX.1 еще не определен.

На рисунке показана иерархия всех классов, предназначенных для работы с файлами, которые были рассмотрены в этой главе.



Файловыми объектами манипулируют процессы, запущенные в операционной системе. Интерфейсы прикладного программирования ОС UNIX и стандарта POSIX, предназначенные для создания процессов и управления ими, рассматриваются в следующей главе.

ГЛАВА 8

Процессы операционной системы UNIX

Процесс — это программа UNIX- или POSIX-системы (например, *a.out*), находящаяся в стадии выполнения. Скажем, shell в UNIX — это процесс, который создается при входе пользователя в систему. Более того, когда пользователь вводит команду *cat foo*, shell создает новый процесс. Согласно терминологии UNIX, это — порожденный процесс, который выполняет команду *cat* от лица пользователя. Процесс, создавший *порожденный процесс* (или процесс-потомок), становится *родительским процессом*. Порожденный процесс наследует от родительского многие атрибуты и планируется ядром UNIX к выполнению независимо от родительского.

Имея возможность создавать несколько параллельно работающих процессов, операционная система может одновременно обслуживать нескольких пользователей и выполнять несколько задач. Таким образом, задача создания процессов и управления ими — краеугольный камень такой многопользовательской, многозадачной операционной системы, как UNIX. Способность процесса создавать в процессе выполнения новые процессы дает следующие преимущества:

1. Любой пользователь имеет право создавать многозадачные приложения.
2. Поскольку порожденный процесс выполняется в собственном виртуальном адресном пространстве, успех или неудача при его выполнении на родительский процесс не влияют. Родительский процесс может после завершения порожденного им процесса запросить статус завершения и статистические данные, относящиеся к периоду выполнения порожденного процесса.

3. Очень часто процессы создают порожденные процессы, которые выполняют новые программы (например, программу *spell*). Это позволяет пользователям писать программы, которые могут путем вызова других программ расширять свои функциональные возможности, не требуя ввода нового исходного кода.

В этой главе рассматриваются структуры данных ядра UNIX, которые поддерживают создание и выполнение процессов, описывается интерфейс системных вызовов, предназначенных для управления процессами, а также приводится ряд примеров, демонстрирующих приемы разработки многозадачных программ в UNIX.

8.1. Поддержка процессов ядром ОС UNIX

Структура данных и механизм выполнения процессов зависят от реализации операционной системы. Рассмотрим структуру данных процесса и поддержку его на уровне операционной системы на примере ОС UNIX System V.

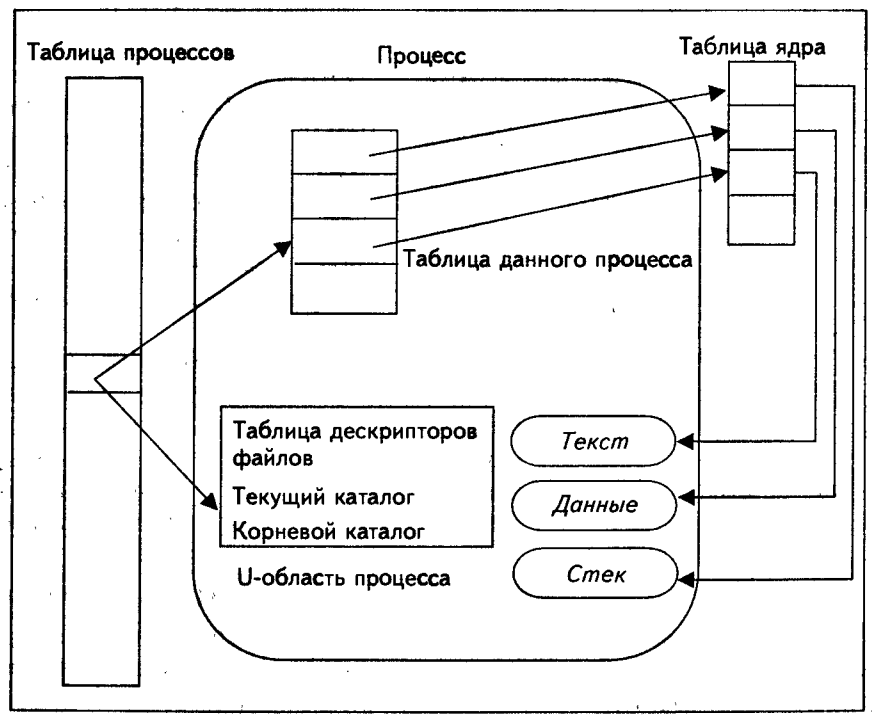


Рис. 8.1. Структура данных UNIX-процесса

Из рис. 8.1 видно, что UNIX-процесс состоит как минимум из сегмента текста, сегмента данных и сегмента стека. Сегмент — это область памяти, которой система управляет как единым целым. Сегмент текста содержит текст программы процесса в формате машинных кодов команд. Сегмент данных содержит статические и глобальные переменные с соответствующими данными. Сегмент стека содержит динамический стек. В стеке хранятся аргументы функций, переменные и адреса возврата всех функций, активных в процессе в каждый данный момент времени.

В ядре UNIX есть таблица процессов, в которой отслеживаются все активные процессы. Некоторые из них инициируются ядром и называются *системными процессами*. Большинство процессов связаны с пользователями, зарегистрированными в системе. Каждый элемент таблицы процессов содержит указатели на сегменты текста, данных, стека и U-область процесса. U-область — это расширение записи в таблице процессов, которое содержит дополнительные данные о процессе, в частности таблицу дескрипторов файлов, номера индексных дескрипторов текущего корневого и рабочего каталогов, набор установленных системой лимитов ресурсов процессов и т.д.

Все процессы в UNIX-системе, кроме самого первого (процесс с ID равным 0), который создается программой начальной загрузки системы, создаются с помощью системного вызова *fork*. После завершения системного вызова *fork* и родительский, и порожденный процессы возобновляют свое выполнение.

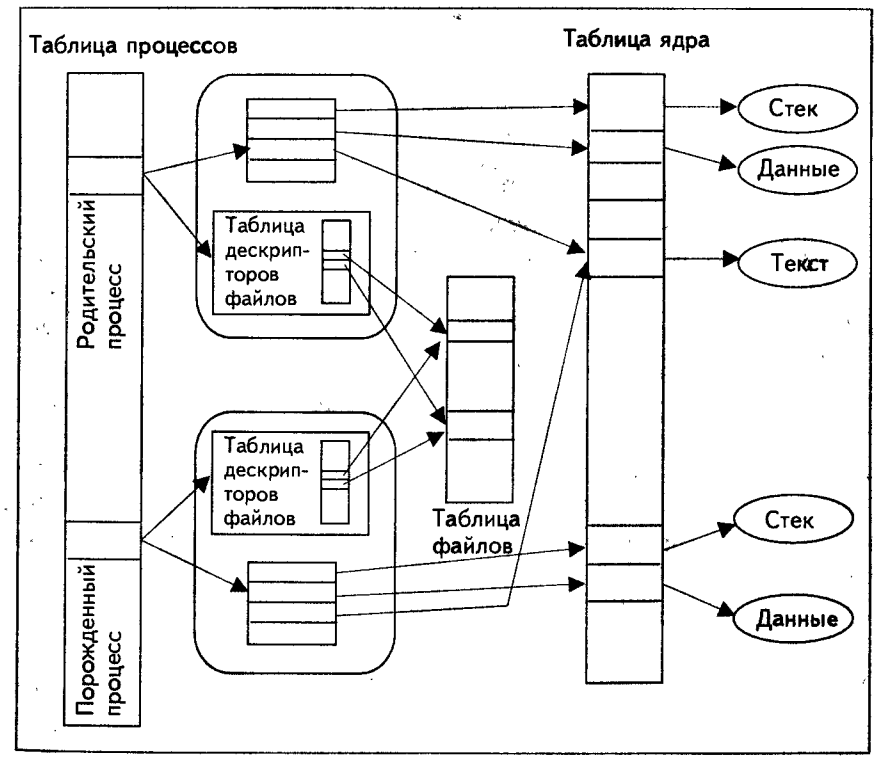


Рис. 8.2. Структура данных родительского и порожденного процессов после завершения системного вызова fork

Судя по рис. 8.2, когда процесс создается функцией *fork*, он получает копии сегментов текста, данных и стека своего родительского процесса. Еще он получает таблицу дескрипторов файлов, которая содержит ссылки на те же открытые файлы, что имеются и у родительского процесса. Это позволяет им совместно пользоваться одним указателем на каждый открытый файл. Кроме того, процессу присваиваются следующие атрибуты, которые либо наследуются от родительского процесса, либо устанавливаются ядром:

- Реальный идентификатор владельца (rUID): идентификатор пользователя, который создал родительский процесс. С помощью этого атрибута ядро следит за тем, кто и какие процессы создает в системе.
- Реальный идентификатор группы (rGID): идентификатор группы, к которой принадлежит пользователь, создавший родительский процесс. С помощью этого атрибута ядро следит за тем, какая группа и какие процессы порождает в системе.
- Эффективный идентификатор пользователя (eUID): обычно совпадает с rUID, за исключением случая, когда у файла, который был выполнен для создания процесса, установлен флаг *set-UID* (с помощью команды или API *chmod*). В этом случае eUID процесса становится равным UID файла. Это позволяет процессу обращаться к файлам и создавать новые файлы, пользуясь такими же привилегиями, как и у владельца выполняемой программы.
- Эффективный идентификатор группы владельца (eGID): обычно совпадает с rGID, за исключением случая, когда у файла, который был выполнен для создания процесса, установлен флаг *set-GID* (с помощью команды или API *chmod*). В этом случае eGID процесса становится равным GID файла. Это позволяет процессу обращаться к файлам и создавать файлы, пользуясь привилегиями группы, к которой относится программный файл.
- *Saved set-UID* и *saved set-GID*: это назначенные eUID и eGID процесса.
- Идентификационный номер группы процесса (PGID) и идентификационный номер сеанса (SID): обозначают группу, которой принадлежит процесс, и сеанс, участником которого он является.
- Дополнительные идентификационные номера группы: набор вспомогательных идентификаторов группы для пользователя, создавшего процесс.
- Текущий каталог: ссылка (номер индексного дескриптора) на рабочий файл-каталог.
- Корневой каталог: ссылка (номер индексного дескриптора) на корневой каталог.
- Обработка сигналов: параметры обработки сигналов. Сигналы рассматриваются в следующей главе.
- Сигнальная маска: маска, которая показывает, какие сигналы подлежат блокированию.

- Значение *umask*: маска, которая используется при создании файлов для установки необходимых прав доступа к ним.
- Значение *nice*: значение приоритета процесса.
- Управляющий терминал: управляющий терминал процесса.

Следующие атрибуты у родительского и порожденного процессов разные:

- Идентификатор процесса (PID): целочисленный идентификатор, уникальный для процесса во всей операционной системе.
- Идентификатор родительского процесса (PPID): идентификационный номер родительского процесса.
- Ждущие сигналы: набор сигналов, ожидающих доставки в родительский процесс.
- Время посылки предупреждающего сигнала: время, через которое процессу будет послан предупреждающий сигнал (устанавливается системным вызовом *alarm*; в порожденном процессе сбрасывается в нуль).
- Блокировки файлов: набор блокировок файлов, созданных родительским процессом, порожденным процессом не наследуется.

После системного вызова *fork* родительский процесс может посредством системного вызова *wait* или *waitpid* приостановить свое выполнение до завершения порожденного процесса или продолжать выполнение независимо от порожденного процесса. В последнем случае родительский процесс может с помощью функции *signal* или *sigaction* (см. главу 9) выявлять или игнорировать завершение порожденного процесса.

Процесс завершает выполнение при помощи системного вызова *_exit*. Аргументом этого вызова является код статуса завершения процесса. По соглашению, нулевой код статуса завершения означает, что процесс завершил выполнение успешно, а ненулевой свидетельствует о неудаче.

С помощью системного вызова *exec* процесс может выполнить другую программу. Если вызов выполняется успешно, ядро заменяет существующие сегменты текста, данных и стека процесса новым набором, который представляет собой новую подлежащую выполнению программу. Тем не менее процесс остается прежним (поскольку идентификатор процесса и идентификатор родительского процесса одинаковы), и его таблица дескрипторов файлов и открытые каталоги остаются в основном теми же (за исключением дескрипторов, у которых флаг *close-on-exec* был установлен системным вызовом *fcntl*,— после выполнения *exec* они будут закрыты). То есть процесс после вызова *exec* можно сравнить с человеком, меняющим работу. Сменив место работы, он имеет то же имя, те же личные данные, но выполняет другие обязанности.

Когда программа, вызванная посредством *exec*, заканчивает выполнение, она завершает процесс. Код статуса завершения этой программы сообщается родителю данного процесса с помощью функции *wait* или *waitpid*.

Функции *fork* и *exec*, как правило, используются вместе для порождения подпроцесса, предназначенного для выполнения другой подпрограммы. Например, shell в UNIX выполняет каждую команду пользователя путем вызова

fork и *exec*, и затребованная команда выполняется в порожденном процессе. Вот преимущества этого метода:

- процесс может создавать несколько других процессов для параллельного выполнения нескольких программ;
- поскольку каждый порожденный процесс выполняется в собственном виртуальном адресном пространстве, статус его выполнения не влияет на родительский процесс.

Два или более связанных процесса (родительский с порожденным или порожденный с порожденным, имеющие одного родителя) могут взаимодействовать с другими процессами путем организации неименованных каналов. Несвязанные процессы могут общаться по именованным каналам или с помощью средств межпроцессного взаимодействия, описанных в главе 10.

8.2. API процессов

8.2.1. Функции fork, vfork

Системный вызов *fork* используется для создания порожденного процесса. Прототип функции *fork* имеет следующий вид:

```
#ifdef _POSIX_SOURCE
#include <sys/stdtypes.h>
#else
#include <sys/types.h>
#endif

pid_t fork ( void )
```

Функция *fork* не принимает аргументов и возвращает значение типа *pid_t* (определяемое в <sys/types.h>). Этот вызов может давать один из следующих результатов:

- Успешное выполнение. Создается порожденный процесс, и функция возвращает идентификатор этого порожденного процесса родительскому. Порожденный процесс получает от *fork* нулевой код возврата.
- Неудачное выполнение. Порожденный процесс не создается, а функция присваивает переменной *errno* код ошибки и возвращает значение -1.

Ниже перечислены основные причины неудачного выполнения *fork* и соответствующие значения *errno*:

Значение *errno*	Причина
ENOMEM	Для создания нового процесса не хватает свободной памяти
EAGAIN	Количество текущих процессов в системе превышает установленное системой ограничение, попытайтесь повторить вызов позже

Необходимо отметить, что существуют устанавливаемые системой ограничения на максимальное количество процессов, создаваемых одним пользователем (CHILD_MAX), и максимальное количество процессов, одновременно существующих во всей системе (MAXPID). Если любое из этих ограничений при вызове *fork* нарушается, то функция возвращает код неудачного завершения. Символы MAXPID и CHILD_MAX определяются соответственно в заголовках <sys/param.h> и <limits.h>. Кроме того, процесс может получить значение CHILD_MAX с помощью функции *sysconf*:

```
int child_max = sysconf (_SC_CHILD_MAX);
```

При успешном вызове *fork* создается порожденный процесс. Структура данных родительского и порожденного процессов после вызова *fork* показана на рисунке 8.2. Как порожденный процесс, так и родительский планируются ядром UNIX для выполнения независимо, а очередность запуска этих процессов зависит от реализации ОС. По завершении вызова *fork* выполнение обоих процессов возобновляется. Возвращаемое значение вызова *fork* используется для того, чтобы определить, является ли процесс порожденным или родительским. Таким образом, родительский и порожденный процессы могут выполнять различные задачи одновременно.

Использование *fork* демонстрируется ниже на примере программы *test_fork.C*. Родительский процесс вызывает *fork* для создания порожденного процесса. Если *fork* возвращает -1, значит, системный вызов не выполнился, и родительский процесс вызывает *perror* для вывода диагностического сообщения в стандартный поток ошибок. Если же *fork* выполнена успешно, то порожденный процесс после своего выполнения выдает на стандартный вывод сообщение *Child process created*. Затем он завершается посредством оператора *return*. Тем временем родительский процесс выводит на экран сообщение *Parent process after fork* и тоже завершает свою работу.

```
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
    switch (fork()) {
    case (pid_t)-1: perror("fork");     /* fork fails */
      break;
    case (pid_t)0: cout << "Child process created\n";
      return 0;
    default: cout << "Parent process after fork\n";
    }
    return 0;
}
```

Выполнение этой программы может дать следующие результаты:

```
% CC -0 test_fork test_fork.C
% test_fork
Child process created
Parent process after fork
```

Альтернативой *fork* является API *vfork*, который имеет такую же сигнатуру, как и *fork*:

```
pid_t vfork (void);
```

Системный вызов *vfork* выполняет ту же функцию, что и *fork*, и возвращает те же возможные значения, что и *fork*. Он применяется и в BSD UNIX, и в UNIX System V.4, но не является стандартным для POSIX.1. Создание *vfork* связано с тем, что многие программы вызывают *exec* (в порожденном процессе) сразу после *fork*. В случае, если ядро не создаст отдельного виртуального адресного пространства для порожденного процесса до выполнения *exec*, системный вызов *vfork* повысит производительность системы. В API *vfork* происходит следующее: после того как функция вызвана, ядро приостанавливает выполнение родительского процесса на то время, пока в виртуальном адресном пространстве этого процесса выполняется порожденный процесс. После того как порожденный процесс вызовет *exec* или *_exit*, выполнение родительского процесса возобновится, а порожденный процесс либо получит свое собственное виртуальное адресное пространство после вызова *exec*, либо завершится через вызов *_exit*.

Использование *vfork* небезопасно, так как если порожденный процесс изменяет данные родительского (например, закрывает файлы или модифицирует переменные) до вызова *exec* или *_exit*, названные изменения сохранятся при возобновлении исполнения родительского процесса. Это может вызвать нарушение нормального функционирования родительского процесса. Кроме того, порожденный процесс не должен вызывать *exit* или возвращаться в любую вызывающую функцию, так как это повлечет за собой соответственно закрытие потоковых файлов родительского процесса или изменение его динамического стека.

В последних системах UNIX (например, System V.4) эффективность использования *fork* повышена. В них порожденный и родительский процессы могут совместно использовать общее виртуальное адресное пространство, пока порожденный процесс не вызовет функцию *exec* или *_exit*. Если родительский или порожденный процесс изменит какие-либо данные в совместно используемом виртуальном адресном пространстве, ядро создаст новые страницы памяти для измененного виртуального адресного пространства. Таким образом, процесс, внесший изменения, будет обращаться к новым страницам памяти с измененными данными, тогда как параллельный процесс будет продолжать обращаться к старым страницам. Такая процедура называется копированием при записи (copy-on-write) и обеспечивает выполнение *fork*, по эффективности сравнимое с *vfork*. Поэтому *vfork* следует использовать только для переноса старых приложений в новые UNIX-системы.

8.2.2. Функция _exit

Системный вызов *_exit* завершает процесс. В частности, этот API вызывает освобождение сегмента данных вызывающего процесса, сегментов стека и U-области и закрытие всех открытых дескрипторов файлов. Однако запись в таблице процессов для этого процесса остается нетронутой, с тем чтобы там регистрировался статус завершения процесса, а также отображалась статистика его выполнения (например, информация об общем времени выполнения, количестве пересланных блоков ввода-вывода данных и т.д.). Теперь процесс называется *зомби-процессом*, так как его выполнение уже не может быть запланировано. Родительский процесс может выбрать хранящиеся в записи таблицы процессов данные посредством системного вызова *wait* или *waitpid*. Эти API освобождают также запись в таблице процессов, относящуюся к порожденному процессу.

Если процесс создает порожденный процесс и заканчивается до завершения последнего, то ядро назначит процесс *init* в качестве управляющего для порожденного процесса (это второй процесс, создаваемый после начальной загрузки ОС UNIX. Его идентификатор всегда равен 1). После завершения порожденного процесса соответствующая запись в таблице процессов будет уничтожена процессом *init*.

Прототип функции *_exit* имеет следующий вид:

```
#include <unistd.h>

void _exit (int exit_code);
```

Целочисленный аргумент функции *_exit* — это код завершения процесса. Родительскому процессу передаются только младшие 8 битов этого кода. Код завершения 0 означает успешное выполнение процесса, а ненулевой код — неудачное выполнение. В некоторых UNIX-системах в заголовке <stdio.h> определяются символические константы EXIT_SUCCESS и EXIT_FAILURE, которые могут быть использованы как аргументы функций *_exit*, соответствующие успешному и неудачному завершению.

Функция *_exit* никогда не выполняется неудачно, поэтому возвращаемого значения для нее не предусмотрено.

Библиотечная С-функция *exit* является оболочкой для *_exit*. В частности, *exit* сначала очищает буферы и закрывает все открытые потоки вызывающего процесса. Затем она вызывает все функции, которые были зарегистрированы посредством функции *atexit* (в порядке, обратном порядку их регистрации), и, наконец, вызывает *_exit* для завершения процесса.

В приведенной ниже программе *test_exit.C* демонстрируется использование функции *_exit*. Когда программа запускается, она объявляет о своем существовании и затем завершается посредством вызова *_exit*. Для обозначения успешного выполнения программы она передает нулевое значение кода завершения.

```
#include <iostream.h>
#include <  unistd.h>
int main()
{
    cout << "Test program for _exit" << endl;
    _exit(0);
    return 0;
}
```

После выполнения этой программы пользователи могут проверить статус ее завершения с помощью переменных *status* (в C-shell) или *$?* (в Bourne-shell). Программа может дать следующие результаты:

```
% CC -o test_exit test_exit.C; test_exit
Test program for _exit
% echo $status
0
```

8.2.3. Функции wait, waitpid

Родительский процесс использует системные вызовы *wait* и *waitpid* для перехода в режим ожидания завершения порожденного процесса и для выборки его статуса завершения (присвоенного порожденным процессом с помощью функции *_exit*). Кроме того, эти вызовы освобождают ячейку таблицы процессов порожденного процесса с тем, чтобы она могла использоваться новым процессом. Прототипы этих функций выглядят так:

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

pid_t  wait  ( int* status_p );
pid_t  waitpid  ( pid_t child_pid, int* status_p, int options )
```

Функция *wait* приостанавливает выполнение родительского процесса до тех пор, пока ему не будет послан сигнал, либо пока один из его порожденных процессов не завершится или не будет остановлен (а его статус еще не будет сообщен). Если порожденный процесс уже завершился или был остановлен до вызова *wait*, функция *wait* немедленно возвратится со статусом завершения порожденного процесса (его значение содержится в аргументе *status_p*), а возвращаемым значением функции будет PID порожденного процесса. Если, однако, родительский процесс не имеет порожденных процессов, завершения которых он ожидает, или был прерван сигналом при выполнении *wait*, функция возвращает значение -1, а переменная *errno* будет содержать код ошибки. Следует отметить, что если родительский процесс создал более одного порожденного, функция *wait* будет ожидать завершения любого из них.

Функция *waitpid* является более универсальной по сравнению с *wait*. Как и *wait*, функция *waitpid* сообщает код завершения и идентификатор порожденного процесса по его завершении. Однако в случае с *waitpid* в вызывающем

процессе можно указать, завершения какого из порожденных процессов следует ожидать. Для этого необходимо аргументу *child_pid* присвоить одно из следующих значений:

Фактическое значение	Завершения чего ожидает функция
Равное ID порожденного процесса	Порожденного процесса с данным идентификатором
-1	Любого порожденного процесса
0	Любого порожденного процесса, принадлежащего к той же группе процессов, что и родительский
Отрицательное значение, но не -1	Любого порожденного процесса, идентификатор группы (GID) которого является абсолютным значением *child_pid*

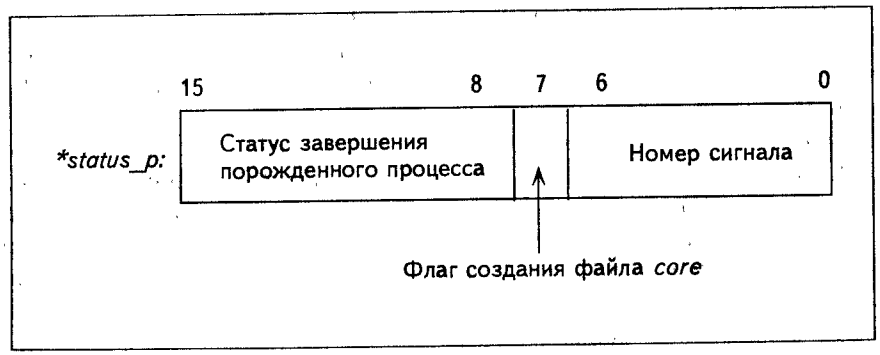
Кроме того, вызывающий процесс может дать функции *waitpid* указание быть блокирующей или неблокирующей, а также ожидать завершения порожденного процесса, приостановленного или не приостановленного механизмом управления заданиями. Эти опции указываются в аргументе *optins*. В частности, если аргумент *options* установлен в значение WNOHANG (этот флаг определен в <sys/wait.h>), вызов будет неблокирующим (то есть функция немедленно возвратит управление, если нет порожденного процесса, отвечающего критериям ожидания). В противном случае вызов будет блокирующим, а родительский процесс будет остановлен, как при вызове *wait*. Далее, если аргумент *options* установлен в значение WUNTRACED, функция также будет ожидать завершения порожденного процесса, остановленного в результате действия механизма управления заданиями (но о статусе которого не было сообщено ранее).

Если фактическое значение аргумента *status_p* вызова *wait* или *waitpid* равно NULL, нет необходимости запрашивать статус завершения порожденного процесса. Однако если его фактическое значение является адресом целочисленной переменной, функция присвоит этой переменной код завершения (указанный в API *_exit*). После этого родительский процесс может проверить этот код завершения с помощью следующих макрокоманд, определенных в заголовке <sys/wait.h>:

Макрокоманда	Возвращаемое значение
WIFEXITED(*status_p)	Ненулевое, если порожденный процесс был завершен посредством вызова *_exit*; в противном случае возвращается нулевое значение
WEXITSTATUS(*status_p)	Код завершения порожденного процесса, присвоенного вызовом *_exit*. Эту макрокоманду следует вызывать лишь в том случае, если WIFEXITED возвращает ненулевое значение
WIFSIGNALED(*status_p)	Ненулевое, если порожденный процесс был завершен по причине прерывания сигналом

Макрокоманда	Возвращаемое значение
WTERMSIG(*status_p)	Номер сигнала, завершившего порожденный процесс. Эту макрокоманду следует вызывать только в том случае, если WIFSIGNALED возвращает ненулевое значение
WIFSTOPPED(*status_p)	Ненулевое, если порожденный процесс был остановлен механизмом управления заданиями
WSTOPSIG(*status_p)	Номер сигнала, завершившего порожденный процесс. Эту макрокоманду следует вызывать лишь тогда, когда WIFSTOPPED возвращает ненулевое значение

В некоторых версиях UNIX, где данные макросы не определены, приведенную выше информацию можно получить непосредственно из **status_p*. В частности, семь младших битов (биты с 0-го по 6-й включительно) **status_p* содержат нуль, если порожденный процесс был завершен с помощью функции *_exit*, или номер сигнала, завершившего процесс. Восьмой бит **status_p* равен 1, если из-за прерывания порожденного процесса сигналом был создан дамп образа процесса (файл *core*). В противном случае он равен нулю. Далее, если порожденный процесс был завершен с помощью API *_exit*, то биты с 8-го по 15-й **status_p* являются кодом завершения порожденного процесса, переданным посредством *_exit*. Следующая схема демонстрирует применение битов данных **status_p*:



В BSD UNIX аргумент *status_p* имеет тип *union wait* (union wait* определяется в <sys/wait.h>). Это объединение целочисленной переменной и набора битовых полей. Битовые поля используются для извлечения такой же информации о статусе, какую получают приведенные выше макросы.

Если возвращаемое значение *wait* или *waitpid* является положительным целочисленным значением, то это — идентификатор порожденного процесса, в противном случае — это *(pid_t)-1*. Последнее означает, что либо ни один из порожденных процессов не отвечает критериям ожидания, либо функция была прервана перехваченным сигналом. Здесь *errno* может быть присвоено одно из следующих значений:

Значение *errno*	Смысл
EINTR	*wait* или *waitpid* возвращает управление вызывающему процессу, так как системный вызов был прерван сигналом
ECHILD	Для *wait* это означает, что вызывающий процесс не имеет порожденного процесса, завершения которого он ожидает. В случае *waitpid* это означает, что либо значение *chld_pid* недопустимо, либо процесс не может находиться в состоянии, определенном значением *options*
EFAULT	Аргумент *status_p* указывает на недопустимый адрес
EINVAL	Значение *options* недопустимо

Как ***wait***, так и ***waitpid*** являются стандартными для POSIX.1. Системный вызов ***waitpid*** отсутствует в BSD UNIX 4.3, System V.3 и их более старых версиях.

На примере приведенной ниже программы ***test_waitpid.C*** демонстрируется использование API ***waitpid***:

```
#include <iostream.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main()
{
   pid_t  child_pid, pid;
   int    status;
   switch (child_pid=fork())
   {
     case (pid_t)-1: perror("fork");     /* fork fails */
       break;
     case (pid_t)0: cout << "Child process created\n";
       _exit(15);              /* terminate child */
     default: cout << "Parent process after fork\n";
       pid = waitpid(child_pid,&status,WUNTRACED);
   }
   if (WIFEXITED(status))
     cerr << child_pid << " exits: " <<WEXITSTATUS(status) << endl;
   else if (WIFSTOPPED(status))
     cerr << child_pid << " stopped by: " <WSTOPSIG(status) << endl;
   else if (WIFSIGNALED(status))
     cerr << child_pid << " killed by: " <<WTERMSIG(status) << endl;
   else perror("waitpid");
   _exit(0);
   return 0;
}
```

Эта простая программа создает порожденный процесс, который подтверждает свое создание, а затем завершается с кодом завершения 15. Тем

временем родительский процесс приостанавливает свое выполнение посредством вызова функции *waitpid*. После завершения порожденного процесса выполнение родительского возобновляется, и его переменным *status* и *child_pid* присваивается код завершения порожденного процесса, а также его идентификатор. Родительский процесс использует макросы, определенные в <sys/wait.h>, для определения статуса выполнения порожденного процесса следующим способом:

- Если WIFEXITED возвращает ненулевое значение, это означает, что порожденный процесс был завершен с использованием вызова *_exit*. Родительский процесс извлекает его код завершения (который в этом примере равен 15) с помощью макрокоманды WEXITSTATUS. Затем он направляет полученное значение в стандартный поток ошибок.
- Если WIFEXITED возвращает нулевое значение, а WIFSTOPPED — ненулевое, то порожденный процесс был остановлен сигналом. Родительский процесс извлекает номер сигнала посредством макрокоманды WSTOPSIG и направляет это значение в стандартный поток ошибок.
- Если и WIFEXITED, и WIFSTOPPED возвращают нулевое, а WIFSIGNALED — ненулевое значение, то порожденный процесс был завершен неперехваченным сигналом. Родительский процесс извлекает номер сигнала с помощью макрокоманды WTERMSIG и направляет это значение в стандартный поток ошибок.
- Если WIFEXITED, WIFSTOPPED и WIFSIGNALED возвращают нулевые значения, это означает, что либо родительский процесс не имеет порожденных, либо вызов *waitpid* был прерван сигналом. Поэтому родительский процесс вызывает функцию *perror* для вывода подробной диагностической информации о причинах неудачи.

Результаты выполнения этой программы могут быть такими:

```
% CC -o test_waitpid test_waitpid.C
% test_waitpid
Parent process after fork
Child process created
1354 exits: 15
%
```

8.2.4. Функция exec

Системный вызов *exec* заставляет вызывающий процесс изменить свой контекст и выполнить другую программу. Есть шесть версий системного вызова *exec*. Все они выполняют одну и ту же функцию, но отличаются друг от друга аргументами.

Вот прототипы функций *exec*:

```
#include <unistd.h>

int execl  (const char* path, const char* arg, ...);
int execlp (const char* file, const char* arg, ...);
int execle (const char* path, const char* arg, ..., const char** env);
int execv  (const char* path, const char** argv, ...);
int execvp (const char* file, const char** argv, ...);
int execve (const char* path, const char** argv, ..., const char** env);
```

Первый аргумент функции является либо полным путевым именем, либо именем файла программы, подлежащей исполнению. Если вызов проходит успешно, данные и команды вызывающего процесса, хранящиеся в памяти, перекрываются данными и командами новой программы. Процесс начинает выполнение новой программы с ее первых строк. После завершения выполнения новой программы код завершения процесса передается обратно родительскому процессу. Следует обратить внимание, что в отличие от *fork*, создающей порожденный процесс, который выполняется независимо от родительского, *exec* лишь изменяет содержание вызывающего процесса для выполнения другой программы.

Вызов *exec* может быть неудачным, если к программе, подлежащей выполнению, нет доступа или если она не имеет разрешения на выполнение. Далее, программа, указанная в первом аргументе вызова *exec*, должна быть исполняемым файлом (имеющим, например, формат *a.out*). В UNIX, однако, можно указать имя сценария shell для вызовов *execlp* и *execvp*, чтобы ядро UNIX для интерпретации сценария shell активизировало Bourne-shell (*/bin/sh*). Так как в POSIX.1 shell не определен, применение *execlp* или *execvp* для выполнения сценариев shell не допустимо. В действительности это не является проблемой, так как пользователи всегда могут активизировать shell вызовом *exec*, указав при этом имя того сценария, который требуется выполнить.

Суффикс *p* в вызовах *execlp* и *execvp* указывает, что если фактическое значение аргумента *file* не начинается с "/", то функции при поиске подлежащего выполнению файла будут использовать переменную среды PATH. Для всех остальных функций *exec* фактическим значением их первого аргумента должно быть путевое имя файла, подлежащего выполнению.

Аргументы *arg* и *argv* являются аргументами также для программы, вызванной функцией *exec*. Они отображаются в переменную *argv* функции *main* новой программы. Для функций *execl*, *execlp* и *execle* аргумент *arg* отображается в *argv[0]*, значение, указанное после *arg*,— в *argv[1]* и т.д. Список аргументов, имеющийся в вызове *exec*, должен завершиться значением NULL, чтобы указать функции, где необходимо прекратить поиск значений аргумента. Для функций *execv*, *execvp* и *execve* аргумент *argv* является массивом (вектором) указателей на символьные строки, где каждая

строка представляет собой одно значение аргумента. Аргумент *argv* отображается непосредственно в переменную *argv* функции *main* новой программы. Так, символ *l* в имени функции *exec* указывает на то, что значения аргумента передаются в виде списка, а символ *v* в этом имени означает, что аргументы передаются в векторном формате.

Следует заметить, что в каждом вызове *exec* необходимо указывать по крайней мере два значения аргументов. Первое значение (*arg* или *argv[0]*) — это имя программы, подлежащей выполнению; оно отображается в *argv[0]* функции *main* новой программы. Второй обязательный аргумент — значение NULL, завершающее список аргументов (для *execl*, *execlp* и *execle*) либо вектор аргументов (для *execv*, *execvp* и *execve*).

Суффикс *e* функции *exec* (*execle* или *execve*) указывает, что последний аргумент (*env*) вызова функции является массивом (вектором) указателей на символьные строки. Здесь каждая строка определяет одну переменную среды и ее значение в формате Bourne-shell:

```
<имя_переменной_среды>=<значение>
```

Последний элемент *env* должен быть NULL, что сигнализирует об окончании списка. В среде, отличной от ANSI C, значение *env* будет присвоено третьему параметру функции *main* выполняемой программы. В среде ANSI C функция *main* может иметь только два аргумента (*argc* и *argv*), а *env* отображается в глобальную переменную *environ* выполняемой программы. При вызове в процессе функций *execl*, *execlp* и *execvp* глобальная переменная *environ* не изменяется (следует отметить, что значение переменной *environ* можно обновить, используя функцию *putenv*).

Если вызов *exec* успешен, сегменты стека, данных и текста первоначального процесса заменяются новыми сегментами, относящимися к выполняемой программе. Однако таблица дескрипторов файлов процесса остается неизменной. Те из файловых дескрипторов, чьи флаги *close-on-exec* были установлены (системным вызовом *fcntl*), будут закрыты перед выполнением новой программы. Во время выполнения программы с помощью API *exec* возможны следующие изменения атрибутов процесса:

- Эффективный идентификатор пользователя: изменяется, если для выполняемой программы установлен бит смены идентификатора пользователя *set-UID*.
- Эффективный идентификатор группы: изменяется, если для выполняемой программы установлен бит смены идентификатора группы *set-GID*.
- Установленный идентификатор пользователя: изменяется, если для выполняемой программы установлен *set-UID*.
- Параметры обработки сигналов: для сигналов, определенных в процессе как перехватываемые, устанавливается режим обработки по умолчанию, когда процесс начинает выполнение новой программы. Определяемые пользователем функции-обработчики сигналов в выполняемой программе отсутствуют.

Большинство программ вызывают *exec* в порожденном процессе, так как выполнение родительского процесса после вызова *exec* желательно продолжать. Однако вызовы *fork* и *exec* реализуются в качестве двух раздельных функций. Это происходит по следующим причинам:

- Реализовать *fork* и *exec* раздельно проще.
- Программа может вызвать *exec* без *fork*, а *fork* без *exec*. Это обеспечивает гибкость в использовании этих функций.
- Многие программы будут производить определенные операции в порожденных процессах, например такие, как переназначение стандартного ввода-вывода в файлы перед вызовом *exec*. Это становится возможным благодаря разделению API *fork* и *exec*.

Использование API *exec* демонстрируется на примере программы *test_exec.C*:

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int System( const char *cmd)        // эмуляция C-функции system
{
   pid_t  pid;
   int    status;
   switch (pid=fork())
   {
      case -1:  return -1;
      case 0:   execl("/bin/sh", "sh", "-c", cmd, 0);
                perror("execl");
                exit(errno);
   }
    if (waitpid(pid,&status,0)==pid && WIFEXITED(status))
       return WEXITSTATUS(status);
    return -1;
}

int main()
{
    int       rc = 0;
    char      buf[256];
    do
   {
       printf("sh> "); fflush(stdout);
       if (!gets(buf)) break;
       rc = System(buf);
   } while (!rc);
    return (rc);
}
```

Приведенная выше программа является простой программой, эмулирующей shell UNIX. Она приглашает пользователя подавать команды shell на стандартный ввод и выполняет каждую команду с помощью функции *System*. Программа завершается, когда пользователь вводит либо признак конца файла ([Ctrl+D]) в командной строке "shell", либо когда статус возврата вызова *System* не равен нулю. Эта программа отличается от shell UNIX тем, что не поддерживает команду *cd* и операции с переменными shell.

Функция *System* эмулирует библиотечную С-функцию *system*. Вот прототип этой функции:

```
int system (const char* cmd);
```

Обе функции вызывают Bourne-shell (*/bin/sh*) для интерпретации и выполнения команды, указанной в аргументе *cmd*. Команда может представлять собой простую команду shell или состоять из серии команд, разделенных точкой с запятой или символами канала (|). Кроме того, этими командами можно задавать переназначение ввода и (или) вывода.

Функция *System* вызывает *fork* для создания порожденного процесса. Порожденный процесс, в свою очередь, вызывает *execlp* для выполнения программы Bourne-shell с аргументами *-c* и *cmd*. Опция *-c* дает Bourne-shell указание интерпретировать и выполнять аргументы *cmd* так, как будто их ввели в командной строке shell. После выполнения *cmd* порожденный процесс завершается, а статус завершения Bourne-shell передается родительскому процессу, который вызывает функцию *System*.

Следует отметить, что функция *System* вызывает *waitpid* специально с целью выявить момент завершения порождаемого процесса. Это важно, так как функция *System* может быть вызвана процессом, породившим другой процесс перед вызовом *System*; таким образом, функция *System* будет ожидать завершения только порожденного ею процесса, а не процессов, которые были созданы вызывающим процессом.

Когда *waitpid* возвращает результат, функция *System* следит за тем, чтобы: возвращенный идентификатор процесса совпадал с идентификатором порожденного ею процесса; порожденный процесс завершался с использованием *_exit*. Если оба условия соблюдаются, функция *System* возвращает статус завершения порожденного процесса. В противном случае она возвращает -1 (неудачное завершение).

Библиотечная функция *system* сходна с функцией *System*, за исключением того, что первая из них производит обработку сигналов и присваивает переменной *errno* код ошибки, когда вызов *waitpid* завершается неудачно.

Пробное выполнение этой программы может дать такие результаты:

```
% CC -o test_exec test_exec.C
% test_exec
sh> date; pwd
Sat May 17 18:09:53 PST 1997
/home/terry/sample
sh> echo "Hello world" | wc > foo; cat foo
1   2  12
sh> ^D
```


8.2.5. Функция pipe

Системный вызов *pipe* создает коммуникационный канал между двумя взаимосвязанными процессами (например, между родительским и порожденным процессами или между двумя процессами-братьями, порожденными одним родительским процессом). В частности, эта функция создает файл канала, который служит в качестве временного буфера и используется для того, чтобы вызывающий процесс мог записывать и считывать данные другого процесса. Файлу канала имя не присваивается, поэтому он называется *неименованным каналом*. Канал освобождается сразу после того, как все процессы закроют файловые дескрипторы, ссылающиеся на этот канал.

```
#include <unistd.h>

int pipe (int fifo[2]);
```

Аргумент *fifo* является массивом, состоящим из двух целых чисел, присваиваемых ему интерфейсом *pipe*. В большинстве систем UNIX канал является однонаправленным, т.е. для чтения данных из канала процесс использует дескриптор файла *fifo[0]*, а для записи данных в канал — другой дескриптор файла, *fifo[1]*. Однако в UNIX System V.4 канал является двунаправленным, поэтому для записи и чтения данных через канал могут использоваться оба дескриптора — *fifo[0]* и *fifo[1]*. Стандарт POSIX.1 поддерживает как традиционные модели каналов ОС UNIX, так и каналы System V.4, не указывая точное назначение дескрипторов каналов. Приложения, которые желательно переносить на все системы UNIX и POSIX, должны использовать каналы так, как будто они однонаправленные.

К хранящимся в канале данным доступ производится последовательно, по алгоритму "первым вошел — первым вышел". Процесс не может использовать функцию *lseek* для произвольного доступа к данным в канале. Данные удаляются из канала сразу же после их считывания.

Общепринятый метод создания коммуникационного канала между процессами с помощью функции *pipe*:

- *Родительский и порожденный процессы*: родительский процесс вызывает *pipe* для создания канала, а затем с помощью *fork* порождает процесс-потомок. Так как порожденный процесс имеет копию дескрипторов файлов родительского, два процесса могут общаться между собой по каналу через свои дескрипторы *fifo[0]* и *fifo[1]*.

- *Порожденные процессы-братья*: родительский вызывает *pipe* для создания канала, затем порождает два или больше процессов-братьев. Порожденные процессы могут сообщаться по каналу посредством своих дескрипторов *fifo[0]* и *fifo[1]*.

Так как буфер, связанный с каналом, имеет конечный размер (PIPE_BUF), то при попытке процесса записать данные в уже заполненный канал ядро заблокирует его до тех пор, пока другой процесс не произведет считывание из канала достаточного количества данных, чтобы заблокированный процесс смог возобновить операцию записи. И наоборот, если канал пуст, а процесс пытается прочитать из него данные, он будет блокироваться до тех пор, пока другой процесс не запишет данные в канал. Эти блокирующие механизмы можно использовать для синхронизации выполнения двух (и более) процессов.

Количество процессов, которые могут параллельно присоединяться к любому концу канала, не ограничено. Однако если два или более процесса записывают в канал данные одновременно, каждый процесс за один раз может записать максимум PIPE_BUF байтов данных. Предположим, процесс (назовем его A) пытается записать X байтов данных в канал, в котором имеется место для Y байтов данных. Если X больше, чем Y, только первые Y байтов данных записываются в канал, и процесс блокируется. Запускается другой процесс (например, B); в это время в канале появляется свободное пространство (благодаря третьему процессу, считывающему данные из канала). Процесс B записывает данные в канал. Затем, когда выполнение процесса A возобновляется, он записывает оставшиеся X-Y байтов данных в канал. В результате данные в канал записываются поочередно двумя процессами. Аналогичным образом, если два (или более) процесса одновременно попытаются прочитать данные из канала, может случиться так, что каждый из них прочитает только часть необходимых данных.

Для предотвращения перечисленных выше недостатков коммуникационный канал обычно делают однонаправленным и устанавливают его только между двумя процессами, чтобы один процесс являлся отправителем, а другой — получателем. Если два процесса, например A и B, нуждаются в двунаправленном коммуникационном канале, они создадут два канала: один для записи данных процесса A в процесс B, другой для осуществления обратной операции.

Если дескриптора файла, связанного с записывающей стороной канала, не существует, то последняя считается "закрытой" и любой процесс, пытающийся считать данные из канала, получает оставшиеся в нем данные. Однако если все данные из канала уже выбраны, процесс, пытающийся продолжать чтение данных из канала, получает признак конца файла (системный вызов *read* возвращает нулевое значение). С другой стороны, если не существует дескриптора файла, связанного с читающей стороной канала, а процесс пытается записать данные в канал, он получает от ядра сигнал SIGPIPE (разорванный канал). Это происходит потому, что данные, записанные в канал, не могут быть выбраны процессом и, следовательно, операция записи считается недопустимой. Процесс, производящий запись, будет "наказан"

этим сигналом (действие этого сигнала по умолчанию — преждевременное завершение процесса).

Системный вызов *pipe* используется shell UNIX для реализации командного канала ("|") с целью соединения стандартного вывода одного процесса со стандартным вводом другого. Он также применяется для реализации библиотечных C-функций *popen* и *pclose*. Описание этих функций приводится в следующем разделе.

В случае успешного завершения *pipe* возвращает 0, а в случае неудачи — -1. Возможные значения, которые этот API присваивает переменной ***errno***, приведены в следующей таблице:

Значение *errno*	**Смысл**
EFAULT	Аргумент *fifo* задан неправильно
ENFILE	Системная таблица файлов заполнена

Как используется API *pipe*, показано на примере следующей программы:

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
  pid_t  child_pid;
  int    fifo[2], status;
  char   buf[80];

  if (pipe(fifo)==-1) perror("pipe"), exit(1);
  switch (child_pid=fork())
  {
     case -1: perror("fork");
              exit(2);
     case 0:  close(fifo[0]);        /* порожденный процесс */
              write(fifo[1],"Child %d executed\n",getpid());
              close(fifo[1]);
              exit(0);
  }
  close(fifo[1]);                    /* родительский процесс */
  while (read(fifo[0],buf,80)) cout << buf << endl;
  close(fifo[0]);
  if (waitpid(child_pid,&status,0)==child_pid && WIFEXITED(status))
     return WEXITSTATUS(status);
  return 3;
}
```

В этой программе показан простой вариант использования *pipe*. Родительский процесс вызывает *pipe* для создания канала. Затем он вызывает *fork* для создания порожденного процесса. Как родительский, так и порожденный процессы имеют доступ к каналу через свои собственные копии переменной *fifo*.

В данном примере порожденный процесс определяется как отправитель сообщения родительскому, записывающий в канал сообщение *Child <child_pid> executed* посредством дескриптора *fifo[1]*. Системный вызов *getpid* возвращает значение идентификатора порожденного процесса. После записи порожденный процесс завершается с помощью *exit* с нулевым кодом возврата.

Родительский процесс определяется как получатель, который читает сообщение порожденного процесса с помощью дескриптора *fifo[0]*. Следует отметить, что родительский процесс сначала закрывает дескриптор *fifo[1]*, а затем входит в цикл для чтения данных из канала. Это необходимо для того, чтобы записывающая сторона канала закрывалась, когда порожденный процесс после завершения записи закроет свой дескриптор *fifo[1]*. После чтения всех сообщений порожденного процесса родительский процесс получает признак конца файла. Если родительский процесс не закроет *fifo[1]* до окончания своего цикла чтения, то он будет заблокирован в системном вызове *read* сразу же после считывания всех данных из канала (записывающая сторона канала, обозначенная дескриптором файла *fifo[1]*, все еще открыта, поэтому признака конца файла в канале не будет).

Как правило, читающий процесс всегда закрывает дескриптор записывающей стороны канала перед чтением данных из канала. Аналогичным образом процесс-отправитель всегда должен закрывать дескриптор записывающей стороны канала после завершения записи данных в канал. Это позволяет читающему процессу выявлять конец файла.

После выхода из цикла чтения родительский процесс вызывает *waitpid* для получения статуса завершения порожденного процесса и завершается либо с кодом завершения порожденного процесса (если порожденный процесс завершился с помощью *_exit*), либо с кодом неудачного завершения 3.

Выполнение этой программы может дать следующий результат:

```
% CC test_pipe.C -o test_pipe;  test_pipe
Child 1234 executed
```

8.2.6. Переназначение ввода-вывода

В ОС UNIX процесс может с помощью библиотечной С-функции *freopen* изменять свои порты стандартного ввода и (или) вывода, переназначая их с консоли на текстовые файлы. Например, приведенные ниже операторы заменяют стандартный вывод процесса на файл *foo*, чтобы оператор *printf* мог записать в этот файл сообщение *Greeting message to foo*:

```
FILE *fptr = freopen("foo", "w", stdout);
printf("Greeting message to foo\n");
```


Аналогичным образом следующие операторы заменяют стандартный ввод процесса на файл *foo*, направляя содержимое всего файла на стандартный вывод:

```
char buf[256];
FILE *fptr = freopen("foo", "r", stdin);
while (gets(buf)) puts(buf);
```

По сути, функция *freopen* при переназначении стандартного ввода и вывода использует системные вызовы *open* и *dup2*. Так, для переназначения стандартного ввода процесса из файла *src_stream* можно сделать следующее:

```
#include <unistd.h>
int fd = open("src_stream", O_RDONLY);
if (fd != -1) dup2(fd, STDIN_FILENO); close(fd);
```

Сначала указанные операторы открывают файл *scr_stream* только для чтения, а дескриптор *fd* обозначает открытый файл. Если вызов *open* проходит успешно (то есть значение *fd* не равно -1), вызывается функция *dup2* — для того, чтобы макрос STDIN_FILENO (который определен в заголовке <unistd.h> и является значением дескриптора файла стандартного ввода) обозначал теперь файл *scr_stream*. Затем дескриптор *fd* отсоединяется от файла посредством системного вызова *close*. В результате этого дескриптор STDIN_FILENO процесса будет теперь обозначать файл *scr_stream*.

Подобные системные вызовы можно использовать и для перенаправления стандартного вывода процесса в файл:

```
#include <unistd.h>
int fd = open("dest_stream", O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (fd != -1) dup2(fd, STDOUT_FILENO); close(fd);
```

В результате действия перечисленных операторов любые данные, посылаемые на стандартный вывод процесса с дескриптором STDOUT _FILENO, будут записываться в файл *dest_file*.

Функция *freopen* может быть реализована следующим способом:

```
FILE* freopen (const char* file_name, const char* mode,
               FILE* old_fstream)
{
    if (strcmp(mode, "r") && strcmp(mode, "w"))
      return NULL;                    /* недопустимое значение mode */
    int fd = open(file_name, *mode=='r' ? O_RDONLY :
                     O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd == -1) return NULL;
    if (!old_fstream) return fdopen(fd, mode);
    fflush(old_fstream);
    int fd2 = dup2(fd, fileno(old_fstream));
    close(fd);
    return (fd2 == -1) ? NULL : old_fstream;
}
```

Если значение аргумента *mode* не равно "*r*" или "*w*", функция возвращает NULL-указатель потока, так как другие режимы доступа функция не поддерживает. Если файл, заданный аргументом *file_name*, невозможно открыть в указанном режиме, функция также возвращает NULL-указатель потока. Если же вызов *open* проходит успешно и аргумент *old_fstream* равен 0, то исходного потока для переназначения нет. В этом случае функция просто преобразует дескриптор файла *fd* в указатель потока (с помощью функции *fdopen*) и возвратит его вызывающему процессу.

Если, однако, значение *old_fstream* не равно NULL, функция вначале сбросит все данные, хранящиеся в буфере ввода-вывода этого потока, посредством вызова функции *fflush*, а затем с помощью *dup2* сделает так, что дескриптор файла, связанный с *old_fstream*, будет обозначать открытый файл. Следует заметить, что использовать здесь *fclose* для очистки буфера и закрытия *old_fstream* не разрешается. Чтобы открыть новый файл, в функции *freopen* нужно повторно использовать структуру FILE, обозначенную указателем *old_fstream*, но *fclose* освобождает эту структуру. Макрокоманда *fileno* определяется в заголовке <stdio.h>. Она возвращает дескриптор файла, связанный с данным указателем потока.

После вызова *dup2* функция закрывает *fd*, так как он больше не нужен, и возвращает либо *old_fstream*, обозначающий теперь новый файл, либо NULL, если вызов *dup2* прошел неудачно.

8.2.6.1. Переназначение ввода-вывода в ОС UNIX

Конструкции переназначения стандартных потоков ввода (<) и вывода (>) shell UNIX можно реализовать с помощью метода, описанного выше. Правда, операция переназначения будет выполнена до того, как порожденный процесс вызовет функцию *exec* для выполнения команды пользователя. В частности, команда shell может быть реализована с помощью такой программы:

```
% sort < foo > results
```

Приведенная ниже программа демонстрирует одну из возможностей переназначения стандартного ввода и вывода:

```
#include <unistd.h>
int main()
{
   int fd, fd2;
   switch ( fork() )
   {
      case -1:  perror("fork"); break;
      case 0:   if (fd=open("foo", O_RDONLY)) == -1 ||
                      (fd=open("results", O_WRONLY |
                       O_CREAT | O_TRUNC, 0644)) == -1 )
                {
                 perror("open");
                 _exit(1);
                }
```



```
         /* переназначить стандартный ввод из foo */
          if ( dup2(fd, STDIN_FILENO) == -1 ) _exit(5);
         /* переназначить стандартный вывод в result */
          if ( dup2(fd2, STDOUT_FILENO) == -1 ) _exit(6);

          close(fd);
          close(fd2);
          execlp("sort", "sort", 0);
          perror("execlp");
          _exit(8);
   }
    return 0;
}
```

Данная программа создает для выполнения команды *sort* порожденный процесс. Этот процесс переназначает свой стандартный ввод на файл *foo*, а стандартный вывод — на файл *results*. Если оба вызова *open* успешны, порожденный процесс вызывает *exec* для выполнения команды *sort*. Так как у команды *sort* нет аргумента, она будет получать данные со стандартного ввода (файл *foo*). Отсортированные данные направляются на стандартный вывод процесса, которым теперь является файл *result*.

Можно также переназначить порты стандартного ввода и (или) вывода родительского процесса еще до вызова *fork* и *exec*. Разница с порожденным процессом заключается в том, что после вызова *fork* родительский процесс либо не использует переназначенный порт, либо устанавливает его в исходное положение (например, в */dev/tty*). Затем его можно использовать таким же образом, как и до вызова функции *fork*. Поэтому в порожденном процессе стандартный ввод и (или) вывод легче переназначать непосредственно перед системным вызовом *exec*.

8.2.6.2. Командные каналы ОС UNIX

Командные каналы shell операционной системы UNIX позволяют одновременно выполнять множество команд таким образом, что стандартный вывод одной команды (указанной слева от символа "|") связывается напрямую со стандартным вводом следующей команды (указанной справа от символа "|"). Так, при получении команды

```
% ls -l | sort -r
```

shell создает два порожденных процесса; один выполняет команду *ls -l*, а другой — команду *sort*.

Данные со стандартного вывода порожденного процесса, выполняющего команду *ls -l*, будут перенаправлены на стандартный ввод порожденного процесса, выполняющего команду *sort*. В результате действия обеих команд будет получен отсортированный перечень содержимого текущего каталога.

Программа *command_pipe.C* демонстрирует возможность одновременного использования двух команд с помощью командного канала:

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

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#define CLOSE_ALL()  close(fifo[0]), close(fifo[1])

int main( int argc, char* argv[])
{
      int    fifo[2];
      pid_t  pid1, pid2;

      if(pipe(fifo)) perror("pipe"), _exit(2);
                                  /* создать командный канал */

      switch (pid1=fork())        /* выполнить команду, указанную
                                     слева от символа канала */
      {
      case -1:  perror("fork"), exit(3);
      case 0:   if (dup2(fifo[1],STDOUT_FILENO)==-1)
           perror("dup2"), exit(4);
           CLOSE_ALL();
           if (execlp("/bin/sh","sh","-c",argv[1],0)==-1)
           perror("execl"), exit(5);
      }

      switch (pid2=fork())        /* выполнить команду справа от
                                     символа канала */
      {
      case -1:  perror("fork"), exit(6);
      case 0:   if (dup2(fifo[0],STDIN_FILENO)==-1)
                     perror("dup2"), exit(7);
                CLOSE_ALL();
                if (execlp("/bin/sh","sh","-c",argv[2],0)==-1)
                     perror("execl"), exit(8);
      }

      CLOSE_ALL();
      if (waitpid(pid1,&fifo[0],0)!=pid1 || waitpid(pid2,&fifo[1],0)!=pid2)
         perror("waitpid");
      return fifo[0]+fifo[1];
}
```

Эта программа принимает в командной строке два аргумента. Каждый аргумент указывает команду, подлежащую выполнению. Если команда содержит аргументы, то она вместе с аргументами должна быть взята в кавычки, чтобы shell передал их как один аргумент программы. Например, в следующей команде аргументом *argv[1]* программы будет *ls -l*, а аргументом *argv[2]* — *sort -r*.

```
% a.out "ls -l" "sort -r"
```

Когда эта программа запускается, она создает между потоками стандартного ввода и вывода двух подлежащих выполнению программ неименованный канал. Если системный вызов *pipe* оказывается неудачным, программа

вызывает *perror* для вывода на экран диагностического сообщения и затем завершает свою работу.

После создания канала программа порождает новый процесс — для выполнения команды, указанной в *argv[1]*. Однако перед вызовом *execl* порожденный процесс переназначает свой поток стандартного вывода на записывающую сторону канала, а затем закрывает свою копию дескриптора канала. Таким образом, при выполнении команды *argv[1]* данные стандартного вывода будут направляться в канал.

Программа создает также другой порожденный процесс, который с помощью вызова *exec* активизирует Bourne-shell. Последний, в свою очередь, выполняет команду, указанную в *argv[2]*. Однако перед вызовом *execl* порожденный процесс переназначает свой поток стандартного ввода на читающую сторону канала и закрывает свою копию дескриптора канала. Поэтому при выполнении *argv[2]* данные стандартного ввода будут получены из канала.

После создания двух порожденных процессов родительский процесс закрывает дескрипторы канала. Это необходимо для того, чтобы при завершении первого порожденного процесса (выполняющего *argv[1]*) отсутствовал дескриптор, связанный с записывающей стороной канала, а второй процесс (выполняющий *argv[2]*) мог прочитать признак конца файла и знал, когда ему нужно самозавершиться. Родительский процесс ожидает завершения двух порожденных процессов и проверяет их статус завершения.

Приведенная выше программа не совместима с POSIX.1, так как Bourne-shell в POSIX.1 не определен. Она, однако, совместима со стандартом POSIX.2, в котором shell и командные каналы определены.

Эта программа может дать следующий результат:

```
% CC -o command_pipe command_pipe.C
% command_pipe  "ls -l"  wc
52   410  3034
% comman_pipe  pwd  cat
/home/book/chapter10
% command_pipe  cat  "sort -r"
Hello world
Bye-Bye
^D
Bye-Bye
Hello world
%
```

Данную программу можно усовершенствовать таким образом, чтобы она использовала произвольное количество команд shell и каналов. В частности, если новая программа запускается командой

```
% command_pipe  "ls -l"  "sort -r"  "wc"  "cat"
```

это аналогично такой команде shell:

```
% ls -l | sort -r | wc | cat
```

Как правило, если команда UNIX содержит *n* символов "|", то shell должен создать *n* каналов (с помощью API *pipe*) и *n+1* порожденных процессов. Так как для каждого вызова канала требуется два файловых дескриптора, а на процесс — максимум OPEN_MAX дескрипторов за один раз, то shell должен использовать свои дескрипторы файлов многократно. Это позволяет системе с неименованными каналами обрабатывать неограниченное количество командных каналов.

На примере программы *command_pipe.C* покажем принцип обработки произвольного числа командных каналов и команд shell:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#define CLOSE(fd)  if (fd < -1) close(fd), fd=-1
static int fifo[2][2] = { -1, -1, -1, -1 }, cur_pipe=0;

int main( int argc, char* argv[])
{
   for (int i=1; i argc; i++)
   {
      if (pipe(fifo[cur_pipe])) perror("pipe"), _exit(2);
      switch (fork())              /* выполнить команду слева от
                                      символа канала */

         case -1:  perror("fork"), exit(3);
         case 0:   if (i > 1)       /* не первая команда */
                   {
                      dup2(fifo[1-cur_pipe][0],STDIN_FILENO);
                      CLOSE(fifo[1-cur_pipe][0]);
                   }
                   if (i < argc-1)    /* не последняя команда */
                   {
                      dup2(fifo[cur_pipe][1],STDOUT_FILENO);
                      CLOSE(fifo[cur_pipe][0]);
                      CLOSE(fifo[cur_pipe][1]);
                   }
                   if (execlp("/bin/sh","sh","-c",argv[i],0)==-1)
                   perror("execl"), exit(5);
      }
      CLOSE(fifo[1-cur_pipe][0]);
      CLOSE(fifo[cur_pipe][1]);
      cur_pipe = 1 - cur_pipe;
   }
   CLOSE(fifo[1-cur_pipe][0]);
   while (waitpid(-1,0,0)) ;
   return 0;
}
```


Эта программа создает уникальный канал между каждыми двумя последовательными командами. Все команды, кроме первой, получают данные стандартного ввода с читающей стороны своего канала (стандартный ввод первой команды осуществляется с консоли). Аналогичным образом все команды, кроме последней, посылают данные своего стандартного вывода на записывающую сторону командного канала (последняя посылает данные стандартного вывода на консоль). Единственным сложным моментом здесь является то, что родительский процесс после создания каждого порожденного процесса закрывает все ненужные дескрипторы каналов. Это делается для того, чтобы родительский процесс не использовал все допустимые дескрипторы файлов и чтобы ненужные дескрипторы не использовались в новом порожденном процессе. Кроме того, родительский процесс должен убедиться в том, что в любой момент времени только один порожденный процесс имеет дескриптор, связанный с одной из сторон канала. Признак конца файла будет передаваться от первого командного процесса к последующему до тех пор, пока его не получат и не завершатся все порожденные процессы.

Пробный запуск этой программы дает следующие результаты:

```
% CC command_pipe2.C
% a.out  ls  sort  wc
150  50  724
% a.out  date
Sun  Apr 19 13:00 PST 1997
% a.out  date  wc
1  6  29
% a.out  pwd  sort  cat
/home/book.chapter11
```

8.2.6.3. Функции popen и pclose

В данном разделе описываются более сложные примеры использования функций *fork*, *exec* и *pipe*. В частности, здесь показано применение этих API для реализации библиотечных С-функций *popen* и *pclose*.

Функцию *popen* можно задействовать для выполнения команды shell в программе пользователя. Прототип функции *popen* выглядит так:

```
FILE*  popen (const char* shell_cmd, const char* mode);
```

Первый аргумент, *shell_cmd*,— это строка символов, содержащая любую команду shell, которую пользователь может задать в Bourne-shell. Рассматриваемая функция вызывает Bourne-shell для интерпретации и выполнения указанной команды.

Второй аргумент, *mode*, равен "*r*" либо "*w*". Он говорит о том, что указатель потока, возвращенный функцией, может быть использован как для чтения данных со стандартного вывода (если *mode* равен "*r*"), так и для их записи на

стандартный ввод (если *mode* равен "*w*") процесса Bourne-shell, выполняющего *shell_cmd*.

Если команда *shell_cmd* не может быть выполнена, *popen* возвращает значение NULL. Возможные причины неудачи: *shell_cmd* — недопустимая команда либо у процесса нет права на ее выполнение.

Функция *pclose* в качестве аргумента принимает указатель потока, возвращенный вызовом функции *popen*. Она закрывает поток, связанный с этим указателем, а затем ожидает завершения соответствующего процесса Bourne-shell. Прототип этой функции выглядит так:

```
int pclose (FILE* fstream);
```

В случае успешного выполнения *pclose* возвращает статус завершения выполняемой команды, а в случае неудачи — -1. Возможная причина неудачи может заключаться в том, что использованное значение *fstream* недопустимо или не определено вызовом *popen*.

Функция *popen* реализуется путем вызова *fork* для создания порожденного процесса, который, в свою очередь, с помощью *exec* вызовет Bourne-shell (*/bin/sh*) для интерпретации и выполнения команды *shell_cmd*. Кроме того, родительский процесс вызывает *pipe*, чтобы установить соединение либо между читаемой стороной канала и стандартным выводом порожденного процесса (если значение mode равно "*r*"), либо между записывающей стороной канала и стандартным вводом порожденного процесса (если значение mode равно "*w*"). Дескриптор файла другого конца канала преобразуется с помощью функции *fdopen* в указатель потока и возвращается в процесс, вызвавший функцию *fdopen*.

Функция *popen* может быть реализована следующим образом:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>
#include <limits.h>

struct sh_rec
{
    pid_t sh_pid;
    FILE* stream;
}   sh_info[OPEN_MAX];
static int  num_sh;

FILE* popen(const char* shell_cmd, const char* mode)
{
    int fifo[2];
```



```
    if ((strcmp(mode,"r") && strcmp(mode,"w")) || pipe(fifo)==-1)
      return 0;
    switch (sh_info[num_sh].sh_pid=fork())
     {
        case -1:   perror("fork"); return 0;
        case 0:    (*mode=='r') ?   dup2(fifo[1], STDOUT_FILENO):
                                    dup2(fifo[-], STDIN_FILENO);
                 close(fifo[0]);
                 close(fifo[1]);
                 execl("/bin/sh", "sh", "-c", shell_cmd, 0);
                 exit(5);
     ]
     if (*mode=='r')
        {
        close(fifo[1]);
        return (sh_info[num_sh++].stream=fdopen(fifo[0], mode));
        ]
     else {
        close(fifo[0]);
        return (sh_info[num_sh++].stream=fdopen(fifo[1], mode);}
}
```

Следует отметить, что идентификатор каждого порожденного процесса и соответствующий указатель потока, полученные из вызова *popen*, записываются в глобальный массив *sh_info*. Массив *sh_info* имеет OPEN_MAX-1 элементов, так как для выделения указателей потоков, обозначающих выполняемые команды shell, процесс может вызвать функцию *popen* максимум OPEN_MAX-1 раз. Данные, хранящиеся в массиве *sh_info*, используются функцией *pclose* следующим образом: после вызова *pclose* находит в массиве *sh_info* элемент, значение *stream* которого совпадает со значением аргумента *fstream*. Если совпадения нет, то значение *fstream* является недопустимым и функция *pclose* возвращает код неудачного завершения -1. Если совпадение есть, функция *pclose* закрывает поток (конец канала), указанный значением *fstream*, а затем ожидает окончания соответствующего порожденного процесса (идентификатор которого задан переменной *sh_pid* в элементе *sh_info*) и возвращает нулевой код успешного завершения.

Функцию *pclose* можно реализовать следующим образом:

```
int pclose (FILE* fstream)
{
    int i, status, rc=-1;
    for (i=0; i < num_sh; i++)
    if (sh_info[i].stream==fstream) break;
    if (i==num_sh) return -1;   /* недопустимое значение fstream */
    fclose(fstream);
    if (waitpid(sh_info[i].sh_pid, &status, 0)==sh_info[i].sh_pid ||
                                        WIFEXITED(status))
      rc = WEXITSTATUS(status);
    for (i++; i < num_sh; i++) sh_info[i-1] = sh_info[i];
    num_sh--;
    return rc;
}
```

Функции *popen* и *pclose* определены в POSIX.2, но не в POSIX.1. Их использование демонстрируется ниже на примере программы *test_popen.C*:

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

int main (int argc, char* argv[])
{
   char buf[256], *mode= (argc>2) ? "w" : "r";
   FILE *fptr;

   if (argc>1 && (fptr = popen(argv[1],mode)))
   {
       switch (*mode)
       {
         case 'r': while (fgets(buf,256,fptr)) fputs(buf,stdout);
                   break;
         case 'w': fprintf(fptr,"%s\n",argv[2]);
       }
       return pclose(fptr);
   }
   return 5;
}
```

Синтаксис вызова этой тест-программы выглядит так:

```
% a.out <shell_cmd> [ <данные, предназначенные для записи в shell_cmd> ]
```

Если вызов осуществляется только с одной командой shell (то есть с одним аргументом), то программа вызывает функцию *popen* для выполнения команды и вывода на консоль данных, поступающих на стандартный вывод выполняемой команды. Если же программа вызывается с двумя аргументами, она с помощью *popen* выполняет указанную в первом аргументе команду, а второй аргумент передает на стандартный ввод выполняемой команды.

Для завершения вызова *popen* программа вызывает функцию *pclose*. Ее пробное выполнение, как правило, дает такие результаты:

```
% CC test_popen.C
% a.out date
Sat Apr 12 21:42:22 PST 1997
% a.out wc "Hello world"
1  2          11
```

8.3. Атрибуты процессов

Описание процессов не будет полным без упоминания различных API, предназначенных для запроса и установки некоторых атрибутов процессов. В этом разделе будут описаны API запроса атрибутов процесса, являющиеся общими для UNIX и POSIX.1. В следующем разделе мы рассмотрим API UNIX и POSIX.1, меняющие атрибуты процессов.


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

pid_t getpid (void);
pid_t getppid (void);
pid_t getpgrp (void);
uid_t getuid (void);
uid_t geteuid (void);
gid_t getgid (void);
gid_t getegid (void);
```

Необходимо отметить, что в старых версиях UNIX все перечисленные API возвращают значения типа *int*. Типы данных *pid_t*, *uid_t* и *gid_t* определяются в заголовке <sys/types.h>.

API *getpid* и *getppid* возвращают идентификаторы текущего и родительского процессов соответственно. Для этих системных вызовов аргументы не требуются.

API *getpgrp* возвращает идентификатор группы вызывающего процесса. Каждый процесс в системах UNIX и POSIX принадлежит к группе процессов. Когда пользователь входит в систему, регистрационный процесс становится лидером сеанса и лидером группы процессов. Идентификатор сеанса и идентификатор группы процесса для лидера сеанса совпадают с PID данного процесса. Если регистрационный процесс создает для выполнения заданий новые порожденные процессы, эти процессы будут принадлежать к тому же сеансу и группе, что и регистрационный процесс. Если регистрационный процесс переводит определенные задания в фоновый режим, то процессу, связанному с каждым фоновым заданием, будет присвоен другой идентификатор группы. Если фоновое задание выполняется более чем одним процессом, то процесс, создавший все остальные процессы с целью выполнения данного фонового задания, станет лидером этой группы процессов.

Системные вызовы *getuid* и *getgid* возвращают соответственно реальный идентификатор владельца и реальный идентификатор группы вызывающего процесса. Реальные идентификаторы группы и владельца являются идентификаторами лица, создавшего этот процесс. Например, когда вы входите в UNIX-систему, реальными идентификаторами группы и владельца регистрационного shell являются соответственно ваш идентификатор группы и ваш идентификатор пользователя. Все порожденные процессы, созданные регистрационным shell, также будут иметь ваши реальные идентификаторы группы и пользователя. Вышеупомянутые реальные идентификаторы используются ядром UNIX для того, чтобы следить за тем, какой пользователь создал процесс в системе.

Системные вызовы *geteuid* и *getegid* возвращают значения атрибутов eUID и eGID вызывающего процесса. Эти идентификаторы используются ядром для определения прав вызывающего процесса на доступ к файлам. Такие

атрибуты присваиваются также идентификаторам группы и владельца для файлов, создаваемых процессом. В нормальных условиях эффективный идентификатор пользователя процесса совпадает с его реальным идентификатором. Однако в случае, когда установлен бит смены идентификатора пользователя *set-UID* выполняемого файла, эффективный идентификатор владельца процесса будет равен идентификатору владельца исполняемого файла. Это дает процессу такие же права доступа, какими обладает пользователь, владеющий исполняемым файлом. Аналогичный механизм применим и к эффективному идентификатору группы. Так, эффективный идентификатор группы процесса будет отличаться от реального идентификатора группы, если установлен бит смены идентификатора группы *set-GID* соответствующего исполняемого файла.

Примером использования флага *set-UID* является команда */bin/passwd*. Эта команда помогает пользователям изменять пароли в файле */etc/passwd*. Так как файл */etc/passwd* обычные пользователи могут только читать, то для того, чтобы записывать данные в файл */etc/passwd*, процесс, созданный в результате выполнения программы */bin/passwd*, должен иметь права привилегированного пользователя. Поэтому идентификатор владельца программы */bin/passwd* является идентификатором привилегированного пользователя, а флаг *set-UID*, определяющий режим доступа к этому файлу, установлен. Процесс, созданный в результате выполнения программы */bin/passwd*, будет иметь эффективный идентификатор привилегированного пользователя и обладать правом модифицирования файла */etc/passwd* в ходе его выполнения.

Флаги *set-UID* и *set-GID* исполняемого файла можно изменить с помощью команды *chmod* или с помощью API *chmod*.

Способ получения атрибутов процесса демонстрируется на примере программы *getproc.C*:

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

int main ( )
{
      cout << "Process PID: " << (int)getpid() << endl;
      cout << "Process PPID: " << (int)getppid() << endl;
      cout << "Process PGID: " << (int)getpgrp() << endl;
      cout << "Process real-UID: " << (int)getuid() << endl;
      cout << "Process real-GID: " << (int)getgid() << endl;
      cout << "Process effective-UID: " << (int)geteuid() << endl;
      cout << "Process effective-GID: " << (int)getegid() << endl;
      return 0;
}
```


Результат выполнения программы может быть следующим*:

```
% CC getproc.C
% a.out
Process PID: 1254
Process PPID: 200
Process PGID: 50
Process real-UID: 751
Process real-GID: 77
Process effective-UID: 205
Process effective-GID: 77
```

8.4. Изменение атрибутов процесса

В этом разделе описаны API POSIX.1 и UNIX, которые изменяют атрибуты процесса. Следует отметить, что некоторые из атрибутов (например, идентификатор процесса и идентификатор родительского процесса) можно запросить, но нельзя изменить, в то время как другие атрибуты (например, идентификатор сеанса процесса) нельзя запросить, но можно изменить.

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

int setsid   (void);
int setpgid  (pid_t pid, pid_t pgid);
int setuid   (uid_t uid);
int seteuid  (uid_t uid);
int setgid   (gid_t gid);
int setegid  (gid_t gid);
```

API *setsid* делает вызывающий процесс лидером нового сеанса и новой группы процессов. Идентификатор сеанса и идентификатор группы вызывающего процесса совпадают с его PID. Процесс будет отсоединен от своего управляющего терминала. Этот API обычно вызывается процессом-демоном после создания последнего, чтобы процесс-демон мог работать независимо от своего родительского процесса. Такой API характерен для POSIX.1.

API *setpgid* делает вызывающий процесс лидером новой группы процессов. Идентификатор группы вызывающего процесса совпадает с его PID. Этот процесс будет единственным членом в новой группе. Вызов рассматриваемого API будет неудачным, если вызывающий процесс уже является лидером сеанса. В UNIX System V API *setpgrp* эквивалентен API *setpgid*.

* Реальные и эффективные идентификаторы не будут совпадать, если для файла *a.out* посредством команды *chmod* установлены биты смены идентификаторов пользователя и группы. — *Прим. ред.*

Если эффективный идентификатор владельца процесса совпадает с идентификатором привилегированного пользователя, системный вызов *setuid* изменяет реальный UID, эффективный UID и установленный UID на значение аргумента *uid*. Если, однако, эффективный идентификатор вызывающего процесса не равен ID привилегированного пользователя, то, если *uid* равен реальному, либо установленному UID, этот API изменяет эффективный идентификатор пользователя на значение *uid*. В противном случае API возвращает код неудачного завершения -1.

Если эффективный идентификатор группы процесса — это идентификатор группы привилегированного пользователя, системный вызов *setgid* определяет реальный идентификатор группы, эффективный идентификатор группы и установленный идентификатор группы как значение аргумента *gid*. Если, однако, идентификатор вызывающего процесса не равен эффективному идентификатору привилегированного пользователя, то, если *gid* равен либо реальному, либо установленному GID, этот API заменяет эффективный идентификатор группы значением *gid*. В противном случае API возвращает код неудачного завершения -1.

API *seteuid* изменяет эффективный идентификатор владельца вызывающего процесса на значение *uid*. Если процесс не обладает правами привилегированного пользователя, значение *uid* должно быть либо реальным идентификатором пользователя, либо сохраненным установленным идентификатором пользователя процесса. С другой стороны, если процесс обладает правами привилегированного пользователя, *uid* может иметь любое значение.

API *setegid* изменяет эффективный идентификатор группы вызывающего процесса на значение *gid*. Если процесс не имеет прав привилегированного пользователя, *gid* должен быть либо реальным идентификатором группы, либо установленным сохраненным идентификатором группы процесса. С другой стороны, если процесс имеет права привилегированного пользователя, *gid* может иметь любое значение.

8.5. Программа minishell

Чтобы показать, как используются API процессов, в данном разделе описывается простая программа эмуляции shell, создающая процессы, которые могут последовательно или параллельно выполнять любое количество UNIX-команд. Это могут быть любые указанные пользователем команды с переназначением ввода и (или) вывода, а также команды в фоновом и (или) приоритетном (foreground) режимах. Единственным недостатком этой программы является то, что она не поддерживает переменные shell и метасимволы.

Программа minishell построена следующим образом. Простая UNIX-команда состоит из имени, необязательных ключей и любого количества аргументов. Кроме того, для каждой команды может быть использовано переназначение ввода-вывода, для соединения различных простых команд можно организовать каналы, а для выполнения команды в фоновом режиме

указать в конце команды символ "&". Ниже приведены примеры команд UNIX, приемлемых для нашей программы:

```
% pwd > foo                  # простая команда с переназначением вывода
% sort-r /etc/passwd &       # команда, выполняемая в фоновом режиме
% cat -n < abc > foo         # команда с переназначением ввода-вывода
% ls -l | sort-r | cat -n    # два канала с тремя командами
% cat foo; date; pwd > zz    # три команды, выполняемые последовательно
% (spell/etc/motd | sort) > xx; date # выполнение двух команд
                                      во вторичном shell
```

Для обработки этих команд создается класс CMD_INFO, в котором каждый из объектов хранит информацию о простой команде. Определение данного класса выглядит так:

```
class CMD_INFO
{
  public:
     char**  argv;       // список команд и аргументов
     char*   infile;     // переназначенный файл стандартного ввода
     char*   outfile     // переназначенный файл стандартного вывода
     int     backgrnd;   // 1, если команда подлежит выполнению в
                         // фоновом режиме
     CMD_INFO* pSubcmd;  // команды, подлежащие запуску во
                         // вторичном shell
     CMD_INFO* Pipe;     // следующая команда после "|"
     CMD_INFO* Next;     // следующая команда после ";"
```

В частности, в переменной *argv* хранятся имя команды, ее ключи и аргументы в виде массива (вектора) указателей на символьные строки. Так, в команде

```
% sort  -r  /etc/motd > foo &
```

переменная *argv* объекта CMD_INFO, выполняющего эту команду, будет выглядеть следующим образом:

```
argv[0] = "sort"
argv[1] = "-r"
argv[2] = "/etc/motd"
argv[3] = 0;
```

В переменных *infile* и *outfile* хранятся имена переназначенных файлов ввода и вывода, которые указаны в команде. Для приведенного выше примера с командой *sort* переменные *infile* и *outfile* объекта CMD_INFO будут содержать такие значения:

```
infile = o;
outfile = "foo"
```

Переменная *backgrnd* указывает приоритетный или фоновый режим выполнения команды. По умолчанию эта переменная равна 0. Это означает, что соответствующую команду следует выполнять в приоритетном режиме.

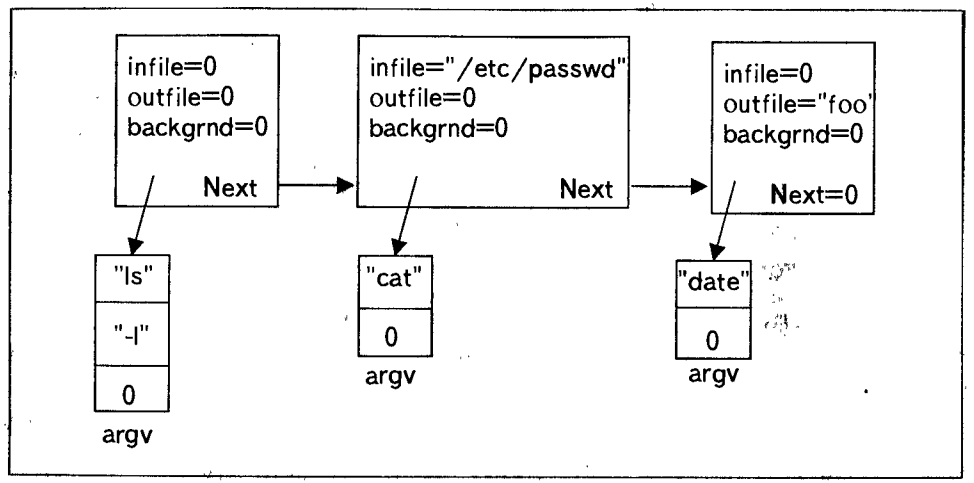
Однако если в командой строке указан символ "&", переменная *backgrnd* этой команды устанавливается в 1. Переменная *backgrnd* объекта CMD_INFO команды *sort* имеет значение:

```
backgrnd = 1;
```

Переменная *Next* используется для привязки другого объекта CMD_INFO, команда которого выполняется тем же процессом shell после команды текущего объекта. Это дает возможность пользователям выполнять в командной строке несколько команд, разделяя их символом ";". Командная строка выглядит так:

```
% ls -l;   cat < /etc/passwd;   date > foo&
```

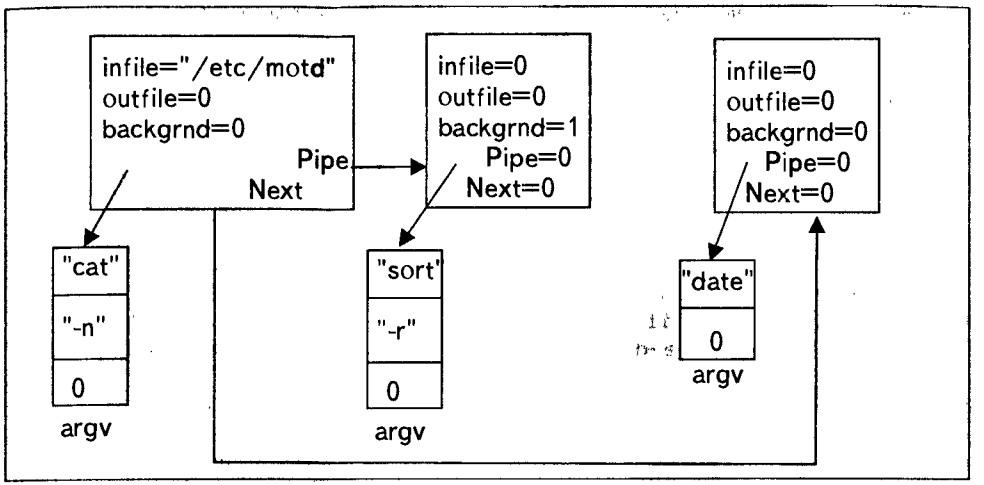
Объекты CMD_INFO, выполняющие эти три команды UNIX, можно представить следующим образом:



Если переменная *Pipe* не равна NULL, то при выполнении команды текущего объекта CMD_INFO данные с ее стандартного вывода посредством канала подаются на стандартный ввод объекта CMD_INFO, указанного переменной *Pipe*. Команды обоих объектов, текущего и указанного переменной *Pipe*, будут выполнены параллельно одним и тем же процессом shell. Таким образом, в команде:

```
% cat -n /etc/motd | sort -r &; date
```

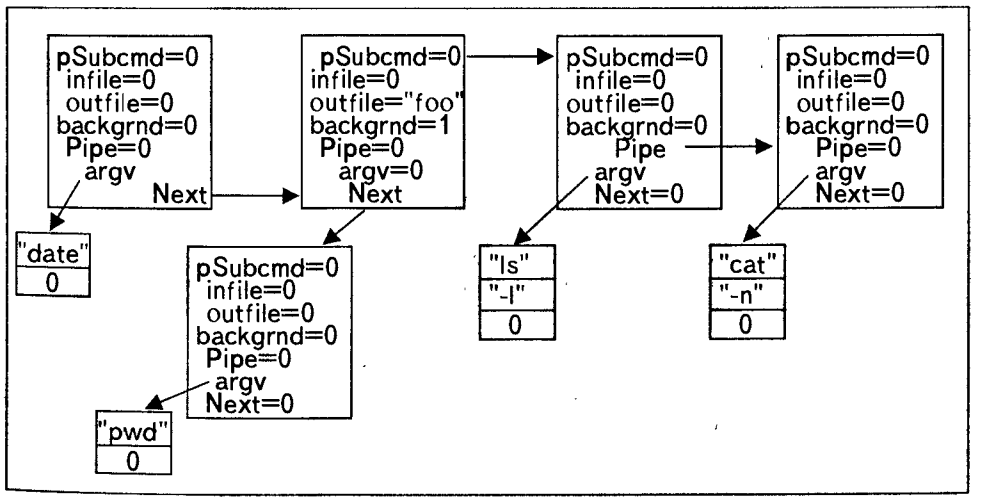
объекты CMD_INFO, выполняющие эти команды, будут такими:



Наконец, переменная *pSubcmd* используется для указания на связный список объектов CMD_INFO, чьи команды выполняются во вторичном shell, отдельном от текущего процесса shell. Если *pSubcmd* не равна NULL, переменная *argv* того же объекта должна равняться 0, так как задачей текущего shell для этой команды является создание вторичного shell, предназначенного для работы с объектами, на которые указывает *pSubcmd*. Так, в команде:

```
% date;   (ls -l | cat -n) > foo &; pwd
```

команды будут выполняться следующими объектами:



В приведенном ниже заголовке *shell.h* объявляются класс CMD_INFO и его функции-члены:

```
/* это файл заголовков shell.h, в котором объявляется класс
CMD_INFO */
#ifndef SHELL_H
#define SHELL_H

#include <iostream.h>
#include <string.h>
#include <assert.h>
#include <malloc.h>

/* проверить переназначения ввода-вывода командных каналов */
#define CHECK_IO(fn) if (fn) { \
    cerr < "Invalid re-direct: " << fn << endl; delete fn; fn = 0; }

class CMD_INFO
{
  public:
     char**    argv;       // список команд и аргументов
     char*     infile;     // переназначенный файл стандартного ввода
     char*     outfile;    // переназначенный файл стандартного вывода
     int       backgrnd;   // 1, если команда должна быть выполнена в
                           // фоновом режиме
     CMD_INFO* pSubcmd;    // команды, выполняемые во вторичном shell
     CMD_INFO* Pipe;       // следующая команда после `|'
     CMD_INFO* Next;       // следующая команда после `;'

     // конструктор
     CMD_INFO()
     {
        argv = 0;
        infile = outfile = 0;
        backgrnd = 0;
        pSubcmd = Pipe = Next = 0;
     };

     // деструктор
     ~CMD_INFO()
     {
        if (infile) delete infile;
        if (outfile) delete outfile;
        for (int i=0; argv && argv(i]; i++) delete argv(i];
        delete argv;
     };

     // добавить одну строку аргументов в argv
     void add_arg( char* str )
     {
        int  len = 1;
        if (!argv)
           argv = (char**)malloc(sizeof(char*)*2);
        else
        {
```



```
           while (argv(len]) len++;
           len++;
           argv = (char**)realloc(argv,sizeof(char*)*(len+1));
        }
        assert(argv[len-1) = strdup(str));
        argv(len] = 0;
     };

   // добавить имя переназначенного файла стандартного ввода или вывода
     void add_iofile( char*& iofile, char* fnm )
      {
        if (iofile)
           cerr << "Multiple in-direct: " << iofile << " vs " << fnm << endl;
        else iofile = fnm;
      };

     // добавить командный канал
     void add_pipe ( CMD_INFO* pCmd )
     {
        if (Pipe)
           Pipe->add_pipe(pCmd);
        else {
           CHECK_IO(outfile);
           CHECK_IO(pCmd->infile);
           Pipe = pCmd;
        }
     };

     // добавить следующую команду
     void add_next( CMD_INFO* pCmd )
     {
        if (Next)
           Next->add_next(pCmd);
        else Next = pCmd;
     };
};

extern void exec_cmd( CMD_INFO *pCmd );
extern "C" int yyparse();
#endif
```

Функция-конструктор CMD_INFO::CMD_INFO инициализирует все переменные вновь созданного объекта нулями.

Функция-деструктор ~CMD_INFO::CMD_INFO освобождает динамическую память, используемую переменными *argv*, *infile* и *outfile* объекта, подлежащего удалению.

Функция CMD_INFO::*add_arg* вызывается для добавления лексем, составляющих команду shell, в переменную *argv* объекта. Эта функция использует функции динамического распределения памяти *malloc* и *realloc* для установки размера *argv* в соответствии с фактическим числом лексем, составляющих команду.

Функция CMD_INFO::*add_iofile* вызывается для добавления имени файла (указанного в аргументе *fnm*) в переменную *infile* или *outfile* объекта (указанную в аргументе *iofile*). Этот файл используется для переназначения стандартного ввода или вывода процесса, который будет создан для выполнения команды объекта.

Функция CMD_INFO::*add_next* добавляет объект CMD_INFO в конец связного списка *Next* объекта. Связный список *Next* задает набор команд, подлежащих выполнению в порядке имеющихся в списке записей.

Функция CMD_INFO::*add_pipe* добавляет объект CMD_INFO в конец связного списка *Pipe* объекта. Связный список *Pipe* задает набор команд, выполняемых параллельно; при этом стандартный вывод одной команды соединяется каналом со стандартным вводом следующей в списке *Pipe* команды. Эта функция следит и за тем, чтобы объект CMD_INFO не передавал данные в канал одновременно с переназначением стандартного вывода в файл. Аналогичным образом объект CMD_INFO не может принимать данные из канала и в то же время перенаправлять свой стандартный ввод в файл.

После определения класса CMD_INFO программа minishell производит синтаксический анализ каждой вводимой строки команды и создает еще один или несколько связных списков объектов CMD_INFO для представления соответствующих команд из входной строки. Анализирующая функция *minishell* состоит из лексического анализатора и синтаксического анализатора, создаваемых с помощью *lex* и *yacc*. В частности, лексический анализатор, распознающий лексемы командной строки, строится из исходного файла *lex* следующим образом:

```
%{/* shell.l: исходный файл лексического анализатора lex для
minishell */
#include <string.h>
#include "shell.h"
#include "y.tab.h"
 %}
%%
;[\t\v]*\n  return '\n';   /* пропустить ";" в конце строки */
^[\t\v]*\n    ;            /* пропустить пустые строки */
^#[^\n]\n     ;            /* пропустить строки комментариев */
#[^\n}*       ;            /* пропустить встроенные комментарии */
[\t\v]     ;               /* пропустить пробельные символы */
[A-Za-z_0-9/,.-]+ { yylval.s = strdup(yytext);
                    return NAME; /* возвратить символьную лексему */
                }
.                           /* односимвольная лексема */
\n                  return yytext[0];
%%

/* программа завершения анализа команд */
int yywrap() { return 1; }
```


Основное назначение лексического анализатора — сбор лексем NAME и специальных символов, таких как "<", ">", "|", "(", ")" и ";", которые образуют одну или более команд shell в каждой вводимой строке.

Лексема NAME состоит из одного или более буквенно-цифровых символов, а также символов "-", ".", "_", "," и "/". Это может быть имя команды shell, ключ или аргумент команды. После обнаружения лексемы NAME лексический анализатор возвращает синтаксическому анализатору символьную строку лексемы (с помощью глобальной переменной *yylval*) и идентификатор лексемы NAME, который определяется в *y.tab.h* (созданный из исходного файла *yacc*). Специальные символы shell "<", ">", "|", "(" и ")" возвращаются синтаксическому анализатору как таковые.

В дополнение к сказанному отметим, что лексический анализатор игнорирует также комментарии (комментарий начинается символом "#" и заканчивается символом новой строки), пробельные символы, пустые строки и необязательный символ ";" в конце входной строки.

Наконец, функция *yywrap* определяется в соответствии с требованиями программы *lex*. Эта функция вызывается, когда лексический анализатор в потоке ввода встречает признак конца файла. Функция дает указание лексическому анализатору прекратить просмотр потока ввода путем возврата единицы, чтобы синтаксический анализатор (а значит, и программа minishell) остановился.

Синтаксический анализатор minishell предполагает, что все вводимые данные состоят из одной или более командных строк. Каждая командная строка заканчивается символом новой строки, и для ее выполнения анализатор вызывает порожденный процесс. Кроме того, командная строка может состоять из одной или более команд, разделенных символом ";". Порожденный процесс выполняет данные команды в порядке их следования. Любая из этих команд может выполняться в фоновом режиме, если в конце ее указан символ "&". Эти синтаксические правила могут быть сформулированы следующим образом:

```
<input_stream> ::= { <cmd_line> '\n' }+
<cmd_line>     ::= <shell {'&'} {';' <shell> {'&'}}+
```

Команда shell может быть простой, с вводом-выводом в канал или сложной:

```
<shell>  ::= <basic> | <pipe> | <complex>
```

Простая команда состоит из имени, необязательных ключей и аргументов, а также обладает возможностью переназначения ввода и (или) вывода. Синтаксис такой команды:

```
<basic>  ::= <cmd> [<switch>]* [<arg>]* {'<' <file>} {'>' <file>}
```

Ниже даны два примера простых команд:

```
ls -l /etc/passwd
cat -n < srcfile > destfile
```

Команда ввода-вывода в канал состоит из набора простых и (или) сложных команд, объединенных командными каналами ("|"). Синтаксис такой команды следующий:

```
<pipe>  ::= [<basic> | <complex>) ['|' <basic> | <complex>]+
```

Ниже приведен пример команды с использованием канала:

```
ls -l /etc/passwd | sort -r | cat -n > foo
```

Сложная команда — это одна или несколько команд shell (простых, с вводом-выводом в канал и сложных, заключенных в скобки и дополнительно сопровождаемых переназначением ввода и вывода). Синтаксис сложной команды:

```
<complex>  ::= '(' <shell> [';' <shell>)* ')' ['<' <file>) [']' <file>]
```

Ниже дан пример сложной команды:

```
( cat < /etc/passwd | sort -r | wc; pwd ) > foo
```

Если объединить все перечисленные синтаксические правила, исходный файл *yacc* для синтаксического анализатора minishell будет выглядеть так:

```
%{ /* shell.y: синтаксический анализатор программы minishell */
#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>
#include "shell.h"
static CMD_INFO *  pCmd = 0;
%)

%union
(
   char*     s;
   CMD_INFO* c;
)
%token <s> NAME

%%
input_stream :   cmd_line  '\n'
            ( exec_cmd($<c>1); )
     | input_stream cmd_line '\n'
            ( exec_cmd($<c>2); )
     | error '\n'
            ( yyerrok; yyclearin;
     ;
cmd_line    :  shell backgrnd
            { $<c>$ = $<c>1; )
```



```
      | cmd_line  ';'  shell backgrnd
            { $<c>1->add_next($<c>3);
              $<c>$ = $<c>1;
            )
      ;

 shell    :  basic
            { $<c>$ = $<c>1; )
      |  complex
            ( $<c>$ = $<c>1; )
      | shell '|' basic
            ( $<c>1->add_pipe($<c>3);
              $<c>$ = $<c>1;
            }
      |  shell  '|'  complex
            ( $<c>1->add_pipe($<c>3);
              $<c>$ = $<c>1;
            )
      ;
basic  :  cmd_spec io_spec
            ( $<c>$ = $<c>1; }
      ;

complex :  '(' cmd_line  ')'
            ( pCmd = new CMD_INFO;
              pCmd->pSubcmd = $<c>2;
            )
        io_spec
            ( $<c>$ = pCmd; )
      ;

cmd_spec   :   NAME
            ( $<c>$ = pCmd = add_vect(0,$<s>1); )
      |   cmd_spec NAME
            { $<c>$ = add_vect($<c>1,$<s>2); }
      ;

io_spec :   /* empty */
      |  io_spec redir
      ;

redir      '<' NAME
            ( pCmd->add_iofile(pCmd->infile, $<s>2); )
      |     '>' NAME
            { pCmd->add_iofile(pCmd->outfile,$<s>2); }
      ;

backgrnd   : /* empty */
      |  '&'
            { pCmd->backgrnd = 1;
      ;
%%
```

```
/* программа выдачи сообщений об ошибках */
void yyerror(char* s) { cerr << s << endl; }

/* добавить команду или аргумент в список */
CMD_INFO *add_vect (CMD_INFO* pCmd, char* str)
{
    int len = 1;
    if (!pCmd) assert(pCmd = new CMD_INFO);
    pCmd->add_arg(str);
    return pCmd;
}
```

В файле *shell.y* синтаксическое правило *cmd_spec* распознает имя команды, необязательные ключи и аргументы. Если это правило "срабатывает", то создается объект CMD_INFO для хранения команды и ее аргументов (с помощью функции *add_vect*), а глобальный указатель *pCmd* устанавливается так, чтобы указывать на этот вновь созданный объект.

С помощью правила *io_spec* собираются имена файлов, связанные с переназначением ввода и (или) вывода, и с применением функции *CMD_INFO::add_file* добавляются к текущему объекту CMD_INFO (на который указывает *pCmd*).

Правило *basic* состоит из правил *cmd_spec* и *io_spec* и служит для распознавания одной элементарной команды, указанной пользователем. Оно передает объект CMD_INFO, созданный правилом *cmd_spec*, в правило *shell*.

Правило *shell* может соответствовать правилам *basic*, *complex* или *pipe*, последнее из которых состоит из набора правил *basic* и (или) *complex*, разделенных лексемой "|". Для правила *pipe* синтаксический анализатор связывает объекты CMD_INFO (с использованием их указателя *CMD_INFO::Pipe*), которые были созданы правилами *basic* и *complex* с помощью функции *CMD_INFO::add_pipe*. Правило *shell* возвращает правилу *cmd_line* либо объект CMD_INFO, созданный правилом *basic* или *complex*, либо связный список этих объектов, обозначенный указателем *Pipe*.

Правило *cmd_line* соответствует одному или нескольким правилам *shell*, каждое из которых может завершаться необязательным символом "&" и отделяться символом ";". Синтаксический анализатор связывает объекты CMD_INFO, возвращенные из правил *shell* (с использованием указателя *CMD_INFO::Next*), в связный список *Next* с помощью функции *CMD_INFO::add_next*. Правило *cmd_line* представляет программе *minishell* одну строку команды и возвращает первый объект CMD_INFO, имеющийся в связном списке *Next* (который строится по правилу *input_stream* или *complex*).

Правило *complex* представляет собой правило *cmd_line*, заключенное в парные круглые скобки. После символа ')' может быть указано переназначение ввода и (или) вывода. Правило *complex* служит для представления набора команд shell, предназначенных для выполнения в отдельном процессе. Синтаксический анализатор создает специальный объект CMD_INFO для обработки данного правила. В этом объекте аргумент *CMD_INFO::argv* равен нулю, указатель *CMD_INFO::pSubcmd* указывает на объект CMD_INFO

(который может быть связным списком *Next*), возвращенный из правила *cmd_line*, а любое переназначение ввода и (или) вывода записывается в переменные *CMD_INFO::infile* и *CMD_INFO::outfile* объекта. Правило *complex* возвращает объект CMD_INFO, созданный им по правилу *shell*.

Наконец, правило *input_stream* состоит из одного или более правил *cmd_line*, каждое из которых завершается символом "\n". Для применения каждого активного правила *cmd_line* синтаксический анализатор вызывает функцию *exec_cmd*, чтобы выполнить команды shell, связанные с объектами CMD_INFO, возвращенными правилом *cmd_line*. Функция *exec_cmd* выглядит следующим образом:

```
/* exec_cmd.C: функции, предназначенные для выполнения одной
   входной строки команды shell */
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#include "shell.h"

/* изменить порт стандартного ввода-вывода процесса */
void chg_io( char* fileName, mode_t mode, int fdesc )
{
   int  fd= open(fileName,mode,0777);
    if (fd==-1)
       perror("open");
   else
   {
      if (dup2(fd,fdesc)==-1) perror("dup2");
       close(fd);
   }
}

/* создать один или несколько командных каналов */
void exec_pipes( CMD_INFO *pCmd )
{
   CMD_INFO   *ptr;
   int     fifo[2][2];
   int     bg=0, first=1, cur_pipe = 0;
   pid_t    pid;
   while (ptr=pCmd)
   {
      pCmd = ptr->Pipe;
      if (pipe(fifo[cur_pipe])==-1)
       {
          perror("pipe"); return;
       }
       switch(fork())
       {
        case -1:    perror("fork");
```

```
            return;
        case 0 : if (!first)   // не первая команда
                 {
                 dup2(fifo[1-cur_pipe][0],0);
                 close(fifo[1-cur_pipe][0]);
                 }
                  else if (ptr->infile)
                  chg_io(ptr->infile,O_RDONLY,0);
                 if (pCmd)     // не последняя команда
                    dup2(fifo[cur_pipe][1],1);
                 else if (ptr->outfile)
                    chg_io(ptr->outfile,
                       O_WRONLY|O_CREAT|O_TRUNC,1);
                 close(fifo[cur_pipe][0]);
                 close(fifo[cur_pipe][1]);
                 execvp(ptr->argv[0],ptr->argv);
                 cerr << "Execute `" << ptr->argv[0] << "` fails\n";
                 exit(4);
        }
        if (!first) close(fifo[1-cur_pipe][0]);
        close(fifo[cur_pipe][1]);
        cur_pipe = 1 - cur_pipe;
        bg = ptr->backgrnd;
        delete ptr;
        first = 0;
    }

    close(fifo[1-cur_pipe][0]);
    while (!bg && (pid=waitpid(-1,0,0))!=-1) ;
}

/* выполнить одну командную строку shell */
void exec_cmd( CMD_INFO *pCmd )
  {
  pid_t     prim_pid, pid;
  CMD_INFO      *ptr = pCmd;

  // создать вторичный shell для обработки одной командной строки
  switch( prim_pid = fork())
  {
     case -1:  perror("fork"); return;
     case 0:   break;
     default:  if (waitpid(prim_pid,0,0)!=prim_pid)  perror("waitpid");
                return;
  }
  while (ptr=pCmd)            // выполнить для каждой команды
  {
     pCmd = ptr->Next;
     if (ptr->Pipe)
        exec_pipes(ptr);      // выполнить команду, связанную с каналом
     else
     {
```



```
        // подпроцесс для выполнения команды
        switch (pid=fork())
        {
           case -1:  perror("fork"); return;
           case 0:   break;
           default:  if (!ptr->backgrnd && waitpid(pid,0,0)!=pid)
                     perror("waitpid");
                     delete ptr;
                     continue;
        }
        if (ptr->infile)
            chg_io(ptr->infile,O_RDONLY,0);
        if (ptr->outfile)
            chg_io(ptr->outfile,O_WRONLY|O_CREAT|O_TRUNC,1);
        if (ptr->argv)
        {
            execvp(ptr->argv[0], ptr->argv);
            cerr << "Execute `" << ptr->argv[0] << "` fails\n";
            exit(999);
        } else
        {
            exec_cmd(ptr->pSubcmd);
            exit(0);
        }
     }
  }
   exit(0);
}
```

Функция *exec_cmd* эмулирует shell, в том смысле, что создает порожденный процесс для выполнения каждой вводимой командной строки. В частности, эта функция вызывается с указателем на объект CMD_INFO, который служит для представления одной или нескольких выполняемых команд. Функция вначале выполняет системный вызов *fork* для создания порожденного процесса, а родительский процесс вызывает *waitpid* для ожидания завершения порожденного процесса. Порожденный процесс, являясь процессом shell, анализирует заданный связный список объектов CMD_INFO и выполняет каждую команду так, как описано ниже. Если в команде используется канал, он вызывает команду *exec_pipe* для выполнения команд, заданных связным списком *Pipe* этого объекта. Если команда является сложной или простой, процесс вторичного shell вызывает *fork*, чтобы создать новый порожденный процесс для выполнения команды следующим образом:

Если в объекте CMD_INFO задано переназначение стандартного ввода и (или) вывода, новый порожденный процесс с помощью функции *chg_io* перенаправляет свой стандартный ввод и (или) вывод на указанные файлы. Если команда является простой, новый порожденный процесс вызывает *exec* для выполнения команды, указанной в *CMD_INFO::argv* объекта. Однако в том случае, когда команда является сложной (*CMD_INFO::pSubcmd* объекта — не NULL), новый порожденный процесс рекурсивно вызывает функцию

exec_cmd для создания отдельного порожденного процесса, который будет выполнять сложные команды в другом контексте.

Процесс вторичного shell ожидает завершения нового порожденного процесса, а затем обрабатывает следующий объект CMD_INFO (если в текущем объекте не установлен флаг *CMD_INFO::background*). Последнее означает, что новый порожденный процесс следует запускать в фоновом режиме, а процесс вторичного shell не будет вызывать *waitpid* для нового порожденного процесса.

Если указатель *CMD_INFO::Pipe* объекта CMD_INFO — не NULL, процесс выполняет команду ввода-вывода в канал, а процесс вторичного shell вызывает функцию *exec_pipe* для создания неименованных каналов и выполнения соответствующих команд новыми порожденными процессами.

Вот функция *main* программы minishell:

```
/* shell.c: основная программа minishell */
#include <iostream.h>
#include <stdio.h>
extern "C" int yyparse();
extern FILE* yyin;

int main(int argc, char* argv[])
{
  if (argc > 1)
    {
      while (--argc > 0)
        if (yyin=fopen(*++argv,"r"))
          yyparse();
      else cerr << "Can't open file: " << *argv << endl;
    }
    else
      yyparse();
  return 0;
}
```

Функция *main* вызовет функцию *yyparse* для синтаксического анализа потока ввода. Функция *yyparse* определяется в файле *y.tab.h*. Поток ввода программы *minishell* может быть связан со стандартным вводом, если у программы нет аргумента командной строки (следовательно, значение *argc* будет равно 1), либо с одним или несколькими текстовыми файлами (сценариями shell), которые специально указываются пользователем во время вызова программы. В последнем случае функция *main* открывает каждый файл сценария и дает функции *yyparse* указание читать данные из этого файла посредством глобального указателя потока *yyin*.

Программа *minishell* может быть скомпилирована следующим образом:

```
% yacc -d shell.y     #создание файлов y.tab.c и y.tab.h
% lex  shell.l         #создание файла lex.yy.c
% CC -o shell shell.C exec_cmd.C lex.yy.c y.tab.c
```


Пробное выполнение этой программы обеспечивает результат, приведенный ниже (команды, вводимые пользователем, даны курсивом, результаты выполнения программы minishell — обычным шрифтом):

```
% shell
date
Sat May 10 11:53:03 PDT 1997
cat -n /etc/motd
   Welcome to T.J.Systems
ls -l | cat -n | sort -r | wc
   10 93 635
pwd; date; ls | wc; ps
/home/terry/test1
Sat May 10 11:53:03 PDT 1997
    9 9  69
PID    TTY     TIME COMD
351    pts/2   0:00 shell
269    pts/2   0:00csh
341    pts/2   0:00shell
356    pts/2   0:01ps
pwd &; (ls -l | cat -n | sort -r | wc) > xyz; cat xyz
/home/terry/test1
   11 103   700
(ls -l | cat -n | sort -r | wc; pwd) &; date
Sat May 10 11:53:03 PDT 1997
   11 103   700
/home/terry/test1
```

8.6. Заключение

В этой главе описаны API операционной системы UNIX и стандарта POSIX, предназначенные для создания процессов, управления ими, обеспечения связи между родительскими и порожденными процессами, запроса и изменения атрибутов процессов. Кроме того, продемонстрированы методы, применяемые для изменения стандартного ввода и вывода процессов, установления командных каналов и выполнения команд shell в пользовательской программе. В конце приведен пример программы minishell, являющейся упрощенной программой shell ОС UNIX, в которой используются все описанные в этой главе API и проиллюстрированы способы их применения.

Важным аспектом управления процессами является обработка сигналов, которая осуществляется в ходе взаимодействии между процессом и ядром операционной системы при управлении асинхронными событиями. Это — основная тема следующей главы.

ГЛАВА 9

Сигналы

Сигналы инициируются событиями и посылаются в процесс для уведомления его о том, что произошло нечто неординарное, требующее определенного действия. Событие может вызываться процессом, пользователем или ядром UNIX. Например, если процесс попытается выполнить математическую операцию "деление на нуль" или разыменовать NULL-указатель, ядро пошлет такому процессу сигнал, который прервет его. Если пользователь нажмет клавишу [Delete] или комбинацию клавиш [Ctrl+C], ядро с помощью сигнала прервет приоритетный (foreground) процесс. Наконец, родительский процесс и порожденные процессы могут посылать друг другу сигналы для синхронизации. Таким образом, сигналы являются программной версией аппаратных прерываний. Точно так же, как в любой системе есть несколько уровней аппаратных прерываний, определены и различные типы сигналов для разных событий, которые могут происходить в UNIX-системе.

Сигналы определяются как флаги, принимающие целочисленные значения. Список сигналов, используемых в UNIX-системах, содержится в заголовке <signal.h>.

Имя сигнала	Ситуация, в которой генерируется сигнал	Создается ли по умолчанию файл core
SIGALRM	Наступление тайм-аута таймера сигналов. Может генерироваться API *alarm()*	Нет
SIGABRT	Аварийное завершение процесса. Может генерироваться API *abort()*	Да
SIGFPE	Недопустимая математическая операция	Да
SIGHUP	Разрыв связи с управляющим терминала	Нет

Имя сигнала	Ситуация, в которой генерируется сигнал	Создается ли по умолчанию файл core
SIGILL	Попытка выполнить недопустимую машинную команду	Да
SIGINT	Прерывание процесса. Обычно генерируется клавишей [Delete] или комбинацией клавиш [Ctrl+C]	Нет
SIGKILL	Уничтожение процесса. Может генерироваться командой *kill -9 <ID_процесса>*	Да
SIGPIPE	Попытка выполнить недопустимую запись в канал	Да
SIGQUIT	Выход из процесса. Обычно генерируется комбинацией клавиш [Ctrl+\]	Да
SIGSEGV	Ошибка сегментации. Может генерироваться при разыменовании NULL-указателя	Да
SIGTERM	Завершение процесса. Может генерироваться командой *kill <ID_процесса>*	Да
SIGUSR1	Зарезервирован и определяется пользователями	Нет
SIGUSR2	Зарезервирован и определяется пользователями	Нет
SIGCHLD	Посылается в родительский процесс при завершении порожденного процесса	Нет
SIGCONT	Возобновление остановленного процесса	Нет
SIGSTOP	Остановка процесса	Нет
SIGTTIN	Остановка фонового процесса, если он пытается прочитать данные со своего управляющего терминала	Нет
SIGSTP	Остановка процесса комбинацией клавиш [Ctrl+Z]	Нет
SIGTTOU	Остановка фонового процесса, если он пытается записать данные на свой управляющий терминал	Нет

Посланный в процесс сигнал обрабатывается. Процесс может отреагировать на ожидающий обработки сигнал одним из трех способов:

- Выполнить действие, предусмотренное по умолчанию. Для большинства сигналов это означает завершение процесса.
- Проигнорировать сигнал. Сигнал отбрасывается и никакого эффекта на процесс-получатель не оказывает.
- Вызвать функцию, определенную пользователем. Эту функцию называют программой-обработчиком сигналов. Когда такая функция вызывается, говорят, что сигнал перехватывается. Если эта функция заканчивает свое выполнение без завершения процесса, то процесс продолжает выполняться с той точки, на которой он был прерван сигналом.

Процесс может создавать для каждого сигнала индивидуальный механизм обработки. Например, одни сигналы он может игнорировать, некоторые

перехватывать, а с остальными выполнять действие, предусмотренное по умолчанию. Более того, в ходе выполнения процесс может изменять принципы обработки определенных сигналов. Например, вначале сигнал может игнорироваться, затем устанавливаться на перехват, а после этого — на выполнение действия по умолчанию. Если процесс-получатель отреагировал на сигнал, то говорят, что сигнал доставлен.

Стандартным действием для большинства сигналов является завершение процесса-получателя (исключение — сигналы SIGCHLD и SIGPWR). Некоторые сигналы генерируют для преждевременно завершенного процесса файл с именем *core*, содержащий образ процесса,— чтобы пользователи могли проанализировать его состояние. Эти сигналы обычно генерируются при наличии в прекращенном процессе неявной программной ошибки. В частности, сигнал SIGSEGV генерируется, когда процесс пытается разыменовать NULL-указатель. Если процесс принимает стандартное действие этого сигнала, то после прекращения процесса создается файл *core*, с помощью которого пользователь может отладить программу.

Большинство сигналов можно игнорировать и перехватывать, за исключением сигналов SIGKILL и SIGSTOP. Сигнал SIGKILL может генерироваться пользователем при работе в shell с помощью команды *kill-9 <ID_процесса>*. Сигнал SIGSTOP останавливает выполнение процесса. Например, если пользователь введет с клавиатуры комбинацию клавиш [Ctrl+Z], ядро пошлет в приоритетный процесс сигнал SIGSTOP, который остановит его. Дополняющий сигнал для SIGSTOP — сигнал SIGCONT, который возобновляет выполнение процесса после остановки. Сигналы SIGSTOP и SIGCONT используются в ОС UNIX для управления выполнением заданий.

Процессу, чтобы он не прерывался во время выполнения ответственных задач, разрешается игнорировать определенные сигналы. Например, когда процесс управления базой данных обновляет файл базы данных, его нельзя прерывать, пока он сам не завершится, иначе этот файл будет поврежден. Поэтому перед обновлением файла данный процесс должен указать, что все основные сигналы прерываний (т.е. SIGINT и SIGTERM) необходимо игнорировать. После обработки файла он должен восстановить порядок обработки сигналов.

Поскольку большинство сигналов выдается в процесс асинхронно, процесс может указать для каждого из них отдельную функцию-обработчик. Такие функции вызываются, когда перехватываются соответствующие сигналы. Одно из типичных для них действий — очистка рабочей среды процесса (в частности, закрытие всех входных и выходных файлов) перед плавным его завершением.

9.1. Поддержка сигналов ядром ОС UNIX

В UNIX System V.3 каждая запись таблицы процессов ядра содержит массив сигнальных флагов, по одному для каждого сигнала, определенного в системе. Когда для процесса генерируется сигнал, ядро устанавливает в

ячейке соответствующей записи таблицы процессов сигнальный флаг для процесса-получателя. Если процесс-получатель находится в режиме ожидания (например, ожидает завершения порожденного процесса или выполняет API *pause*), ядро "будит" его и ставит в очередь на выполнение. Если процесс-получатель работает, ядро проверяет U-область этого процесса, которая содержит массив спецификаций обработки сигналов, где каждый элемент массива соответствует сигналу, определенному в системе. С помощью этого массива ядро определяет, каким образом процесс будет реагировать на ожидающий обработки сигнал. Если элемент массива, соответствующий данному сигналу, содержит нулевое значение, процесс выполняет действие, предусмотренное по умолчанию. Если элемент массива содержит 1, процесс игнорирует сигнал и ядро отбрасывает его. Наконец, если элемент массива содержит другое значение, оно используется как указатель на функцию-обработчик сигналов, определенную пользователем. Ядро настраивает процесс на немедленное выполнение этой функции, и если функция-обработчик не завершает процесс, то он возвращается в ту точку, где был получен сигнал (или в другую точку, куда обработчик сигнала передает управление).

Для случая, когда обработки процессом ожидают разные сигналы, порядок передачи их в процесс-получатель не определен. Если обработки ждут несколько экземпляров одного сигнала, то решение вопроса о том, сколько экземпляров будет доставлено в процесс — один или все, зависит от реализации ОС. В UNIX System V.3 каждый сигнальный флаг в ячейке таблицы процессов регистрирует только наличие ожидающего сигнала; определить же общее количество сигналов невозможно.

Способ обработки перехваченных сигналов, который принят в UNIX System V.2 и более ранних версиях, был раскритикован как ненадежный. Поэтому в BSD UNIX 4.2 (а также более поздних версиях) и POSIX.1 для обработки перехваченных сигналов используются другие механизмы.

В частности, в UNIX System V.2 и более ранних версиях принят порядок, согласно которому при перехвате сигнала ядро сначала деактивизирует обработчик данного сигнала (в U-области процесса-получателя), а затем вызывает заданную для этого сигнала пользовательскую функцию обработки. Таким образом, если в процесс из разных точек посылается множество экземпляров сигнала, процесс перехватывает только первый экземпляр. Все последующие экземпляры сигнала обрабатываются методом, принятым по умолчанию.

Для того чтобы процесс непрерывно перехватывал множество экземпляров сигнала, он должен при каждом перехвате реинсталлировать функцию-обработчик. При этом, однако, все равно нет гарантии того, что процесс перехватит все экземпляры: с момента вызова обработчика перехваченного сигнала X до момента восстановления метода обработки он находится в режиме принятия стандартного действия, предусмотренного для сигнала X. Если за этот промежуток времени в процесс доставляется еще один экземпляр сигнала X, процессу приходится обрабатывать его по стандартному методу. Возникает состояние состязания, при котором два события наступают одновременно, и предсказать, какое из них "сработает" первым, невозможно.

В BSD UNIX 4.2 (а также более поздних версиях) и POSIX.1 для повышения надежности обработки сигналов принят иной подход: когда сигнал перехватывается, ядро не деактивизирует функцию-обработчик, поэтому процессу не нужно восстанавливать метод обработки сигнала. Далее, ядро блокирует дальнейшую доставку этого же сигнала в процесс до тех пор, пока функция-обработчик не закончит выполнение. Появляется гарантия того, что она не будет вызываться рекурсивно для множества экземпляров одного сигнала. В UNIX System V.3 был введен API *sigset*, который работает более надежно, чем API *signal*.

В UNIX System V.4 принят метод обработки сигналов, определенный в POSIX.1. Тем не менее пользователи могут дать ядру указание использовать метод обработки, принятый в System V.2, отдельно по каждому сигналу. Это делается с помощью API *signal*.

9.2. Функция signal

Все UNIX-системы и язык программирования ANSI C поддерживают API *signal*, с помощью которого можно определять методы обработки отдельных сигналов. Прототип функции этого API выглядит следующим образом:

```
#include <signal.h>

void (*signal (int signal_num, void (*handler)(int)))(int)
```

Аргументами данного API являются *signal_num* — идентификатор сигнала (например, SIGINT, SIGTERM и т.д.), определенный в заголовке <signal.h>, и *handler* — указатель на определенную пользователем функцию-обработчик сигнала. Эта функция должна принимать целочисленный аргумент. Никаких значений она не возвращает.

В приведенном ниже примере производится попытка перехвата сигнала SIGTERM, игнорируется сигнал SIGINT, а по сигналу SIGSEGV выполняется действие, предусмотренное по умолчанию. API *pause* приостанавливает выполнение вызывающего процесса до тех пор, пока он не будет прерван каким-либо сигналом и соответствующий обработчик сигнала не возвратит результат:

```
#include <iostream.h>
#include <signal.h>
/* Функция-обработчик сигналов */
void catch_sig( int sig_num )
{
   signal(sig_num, catch_sig);
   cout << "catch_sig:" << sig_num << endl;
}
```

```
/* главная функция */
int main()
{
   signal(SIGTERM, catch_sig);
   signal(SIGINT, SIG_IGN);
   signal(SIGSEGV, SIG_DFL);
   pause();                          /* ожидание сигнала */
}
```

SIG_IGN и SIG_DFL — это макросы, определенные в заголовке <signal.h>:

```
#define SIG_DFL     void (*)(int)0
#define SIG_IGN     void (*)(int)1
```

SIG_IGN задает сигнал, который будет проигнорирован. Будучи посланным в процесс, такой сигнал просто отбрасывается, а сам процесс не прерывается.

SIG_DFL сигнализирует, что нужно выполнить действие по умолчанию.

Возвращаемое значение API *signal* — указатель на предыдущий обработчик данного сигнала. Его можно использовать для восстановления обработчика сигнала:

```
#include <signal.h>
int main()
{
  void (*old_handler)(int) = signal(SIGINT, SIG_IGN);
  /*выполнить важнейшие операции обработки*/
  signal(SIGINT, old_handler); /* восстановить предыдущий обработчик
                                  сигнала */
}
```

API *signal* не является стандартным для POSIX.1. При этом он, однако, определен в ANSI C и имеется во всех UNIX-системах. Так как поведение API *signal* в System V.2 и более ранних версиях отличается от поведения в системах BSD и POSIX.1, то использовать его в переносимых приложениях не рекомендуется. В BSD UNIX и POSIX.1 определен новый набор интерфейсов прикладного программирования, предназначенных для манипулирования сигналами. Эти интерфейсы присутствуют во всех системах UNIX и системах, соответствующих стандарту POSIX.1. Описаны они в следующих двух разделах.

UNIX System V.3 и System V.4 поддерживают API *sigset*, имеющий тот же прототип и назначение, что и *signal*:

```
#include <signal.h>

void (*sigset (int signal_num, void (*handler)(int)))(int)
```

Аргументы и возвращаемое значение API *sigset* — те же, что и у *signal*. Обе функции задают методы обработки для любого указанного в вызове сигнала.

Однако API *signal* в отличии от API *sigset* является недостаточно надежным (см. раздел 9.1). Если обработчик сигнала устанавливается с помощью API *sigset*, то при поступлении нескольких экземпляров этого сигнала обрабатывается один из них, а остальные блокируются. Кроме того, режим обработки сигнала при вызове в SIG_DFL не сбрасывается.

9.3. Сигнальная маска

Каждый процесс в UNIX-системе (BSD 4.2 и выше, а также System V.4) и в POSIX.1 имеет сигнальную маску, которая определяет, какие сигналы, из посылаемых процессу блокируются. Разблокирование и обработка блокированного сигнала выполняются процессом-получателем. Если сигнал задан как игнорируемый и блокируемый, то решение вопроса о том, будет ли он при передаче в процесс отброшен или переведен в режим ожидания, зависит от реализации ОС.

При создании процесс наследует сигнальную маску своего родителя, но ни один сигнал, ожидающий обработки родительским процессом, к порожденному процессу не пропускается. Процесс может запрашивать и создавать сигнальную маску с помощью API *sigprocmask*:

```
#include <signal.h>

int sigprocmask (int cmd, const sigset_t* new_mask, sigset_t* old_mask)
```

Аргумент *new_mask* определяет набор сигналов, которые будут добавлены в сигнальную маску вызывающего процесса или удалены из нее, а аргумент *cmd* указывает, как значение *new_mask* должно быть использовано данным API. Ниже перечислены возможные значения аргумента *cmd* и варианты использования аргумента *new_mask*.

Значение *cmd*	Выполняемые действия
SIG_SETMASK	Заменяет сигнальную маску вызывающего процесса значением, указанным в аргументе *new_mask*
SIG_BLOCK	Добавляет сигналы, указанные в аргументе *new_mask*, в сигнальную маску вызывающего процесса
SIG_UNBLOCK	Удаляет из сигнальной маски вызывающего процесса сигналы, указанные в аргументе *new_mask*

Если фактическим аргументом *new_mask* является NULL-указатель, то аргумент *cmd* игнорируется и сигнальная маска текущего процесса не изменяется.

Аргумент *old_mask* — это адрес переменной типа *sigset_t*, которой присваивается исходная сигнальная маска вызывающего процесса до вызова

sigprocmask. Если фактическим аргументом *old_mask* является NULL-указатель, то предыдущая сигнальная маска не возвращается.

В случае успешного завершения вызов *sigprocmask* возвращает 0, в противном случае возвращает -1. Причиной неудачи может стать неверное задание адресов в аргументах *new_mask* и (или) *old_mask*.

Тип данных *sigset_t* определяется в заголовке <signal.h>. Данные этого типа представляют собой набор битов, каждый из которых является флагом, соответствующим одному определенному в данной системе сигналу.

В BSD UNIX и POSIX.1 определен набор API, известных как семейство функций *sigsetops*, которые устанавливают, сбрасывают и запрашивают флаги сигналов в переменной типа *sigset_t*:

```
#include <signal.h>

int sigemptyset ( sigset_t* sigmask );
int sigaddset ( sigset_t* sigmask, const int signal_num );
int sigdelset ( sigset_t* sigmask, const int signal_num );
int sigfillset ( sigset_t* sigmask );
int sigismember ( const sigset_t* sigmask, const int signal_num);
```

Производимые с помощью этих API действия: *sigemptyset* сбрасывает все сигнальные флаги в маске, указанной в аргументе *sigmask*; *sigaddset* устанавливает флаг, соответствующий сигналу *signal_num*, в маске *sigmask*; *sigdelset* очищает флаг, соответствующий сигналу *signal_num*, в маске *sigmask*; *sigfillset* устанавливает все флаги сигналов в маске *sigmask*.

В случае успешного выполнения вызовы *sigemptyset*, *sigaddset*, *sigdelset*, *sigfillset* возвращают 0, в случае неудачи — -1. Возможной причиной неудачи могут быть неверно заданные аргументы *sigmask* и (или) *signal_num*.

API *sigismember* возвращает 1, если в маске *sigmask* установлен флаг, соответствующий сигналу *signal_num*. В случае неудачи возвращается 0.

В следующем примере проверяется наличие сигнала SIGINT в сигнальной маске процесса. Если такового там нет, он будет добавлен в маску. Затем программа удаляет сигнал SIGSEGV из сигнальной маски процесса:

```
#include <stdio.h>
#include <signal.h>
int main()
{
   sigset_t   sigmask;
   sigemptyset(&sigmask);                /*инициализировать набор*/

if (sigprocmask(0, 0, &sigmask)== -1)    /*получить сигнальную маску*/
{
      perror("sigprocmask");
      exit(1);
}
else sigaddset(&sigmask, SIGINT);        /*установить флаг SIGINT*/
```



```
   sigdelset(&sigmask, SIGSEGV);   /*убрать флаг SIGSEGV*/
   if (sigprocmask(SIG_SETMASK, &sigmask, 0)== -1)
      perror("sigprocmask");       /*установить новую сигнальную маску*/
}
```

Если один или несколько сигналов, ожидающих обработки, будут разблокированы посредством API *sigprocmask*, то до возврата в вызывающий процесс будут применяться методы обработки сигналов, действующие во время вызова *sigprocmask*. Если процесса ждут несколько экземпляров одного сигнала, то решение вопроса о том, сколько экземпляров будет доставлено в процесс — один или все, зависит от реализации ОС.

Процесс может запросить перечень сигналов, ожидающих его, с помощью API *sigpending*:

```
#include <signal.h>

int sigpending ( sigset_t* sigmask );
```

Аргумент *sigmask* этого API является переменной типа *sigset_t*. Этой переменной присваивается набор сигналов, ожидающих вызывающий процесс. В случае успешного выполнения этот API возвращает нуль, а в случае неудачи — -1.

API *sigpending* определяет наличие сигналов, ожидающих данный процесс, и задает специальные методы обработки этих сигналов до того, как процесс вызовет API *sigprocmask* для их разблокирования.

В нашем примере на консоль выдается сообщение о том, ожидает ли обработки сигнал SIGTERM:

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

int main()
{
    sigset_t  sigmask;
    sigemptyset(&sigmask);
    if (sigpending(&sigmask)== -1)
       perror("sigpending");
    else cout << "SIGTERM signal is:"
         << (sigismember(&sigmask,SIGTERM) ? "Set" : "No Set")
         << endl;
}
```

Следует отметить, что помимо перечисленных в UNIX System V.3 и System V.4 в качестве упрощенных средств манипулирования сигнальными масками поддерживаются следующие API:

```
#include <signal.h>

int sighold   ( int signal_num );
int sigrelse  ( int signal_num );
int sigignore ( int signal_num );
int sigpause  ( int signal_num );
```

API *sighold* добавляет заданный сигнал *signal_num* в сигнальную маску вызывающего процесса. Это то же самое, что использование API *sigset* с аргументом SIG_HOLD:

```
sigset (<signal_num>,SIG_HOLD);
```

API *sigrelse* удаляет заданный сигнал *signal_num* из сигнальной маски вызывающего процесса; API *sigignore* устанавливает метод обработки заданного сигнала *signal_num* в значение SIG_DFL; API *sigpause* удаляет заданный сигнал *signal_num* из сигнальной маски вызывающего процесса и приостанавливает процесс до тех пор, пока он не будет возобновлен сигналом *signal_num*.

9.4. Функция sigaction

Интерфейс прикладного программирования *sigaction* заменяет API *signal* в последних версиях ОС UNIX и POSIX.1. Подобно API *signal*, он вызывается процессом для задания метода обработки каждого сигнала, с которым собирается работать. Оба интерфейса прикладного программирования возвращают указатель на предыдущий метод обработки данного сигнала. Кроме того, API *sigaction* позволяет определить дополнительные сигналы, которые также будут блокироваться при обработке сигнала *signal_num*

Прототип API *sigaction* выглядит таким образом:

```
#include <signal.h>

int sigaction ( int signal_num, struct sigaction* action,
                                struct sigaction* old_action);
```

Тип данных *struct sigaction* определяется в заголовке <signal.h>:

```
struct sigaction
{
    void      (*sa_handler)(int);
    sigset_t  sa_mask;
    int       sa_flag;
};
```


Поле *sa_handler* соответствует второму аргументу функции *signal*. В него может быть занесено значение SIG_IGN, SIG_DFL или определенная пользователем функция-обработчик сигнала. Поле *sa_mask* наряду с сигналами, указанными в данный момент в сигнальной маске процесса, и сигналом *signal_num* задает дополнительные сигналы, которые процесс будет блокировать при обработке сигнала *signal_num*.

Аргумент *signal_num* показывает, какое действие по обработке сигнала определено в аргументе *action*. Предыдущий метод обработки сигнала для *signal_num* будет возвращен посредством аргумента *old_action*, если это не NULL-указатель. Если аргумент *action* является NULL-указателем, то метод обработки сигнала вызывающего процесса для *signal_num* останется прежним.

В приведенной ниже программе *sigaction.C* показан пример использования функции *sigaction*:

```
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void callme()
{
     cout << "catch signal" << endl;
}

int main()
{
      sigset_t  sigmask;
     struct sigaction  action, old_action;
     sigemptyset(&sigmask);
     if (sigaddset( &sigmask, SIGTERM)==-1 ||
        sigprocmask(SIG_SETMASK, &sigmask, 0)==-1)
        perror("set signal mask");
      sigemptyset(&action.sa_mask);
      sigaddset(&action.sa_mask,SIGSEGV);
#ifdef SOLARIS_25
     action.sa_handler = (void (*)(int))callme;
#else
     action.sa_handler = callme;
#endif
      action.sa_flags = 0;
     if (sigaction(SIGILL,&action,&old_action)==-1)
        perror( "sigaction");
     pause();                    /* ожидать прерывания сигналом */
     return 0;
}
```

В этом примере сигнальная маска процесса содержит сигнал SIGTERM. Затем процессом определяется обработчик для сигнала SIGINT. Указывается, что сигнал SIGSEGV должен быть блокирован при обработке процессом сигнала SIGINT. Затем процесс приостанавливает свое выполнение с помощью API *pause*.

Пробное выполнение программы дает такой результат:

```
% CC sigaction.C -o sigaction
% sigaction
[1]  495
% kill -INT 495
catch signal
[1]  Interrupt   sigaction
```

Если в процессе генерируется сигнал SIGINT, то ядро сначала устанавливает сигнальную маску процесса на блокирование сигналов SIGTERM, SIGINT и SIGSEGV. Потом оно настраивает процесс на выполнение функции-обработчика сигнала *callme*. Когда функция *callme* возвращает результат, сигнальная маска процесса восстанавливается и содержит только сигнал SIGTERM, а процесс продолжает перехватывать сигнал SIGILL.

Поле *sa_flag* в структуре *struct sigaction* используется для задания специальных методов обработки определенных сигналов. В стандарте POSIX.1 установлено только два значения этого поля: 0 или SA_NOCHLDSTOP. Флаг NOCHLDSTOP — это целое число, определенное в заголовке <signal.h>. Данный флаг может использоваться, когда аргумент *signal_num* имеет значение SIGCHLD. Действие флага SA_NOCHLDSTOP заключается в том, что ядро посылает сигнал SIGCHLD в процесс тогда, когда порожденный им процесс завершен, а не тогда, когда он остановлен. С другой стороны, если значение *sa_flag* в вызове *sigaction* равно 0, то ядро посылает сигнал SIGCHLD в вызывающий процесс вне зависимости от того, завершен порожденный процесс или остановлен.

В ОС UNIX System V.4 определены дополнительные флаги для поля *sa_flag* структуры *struct sigaction*. Эти флаги могут использоваться для указания метода обработки сигнала в стиле UNIX System V.3.

Значение *sa_flag*	Влияние на обработку аргумента *signal_num*
SA_RESETHAND	Если *signal_num* перехвачен, то *sa_handler* устанавливается в SIGDFL до того, как вызывается функция обработчика сигнала, и *signal_num* не добавляется в сигнальную маску процесса при выполнении функции-обработчика сигнала
SA_RESTART	Если сигнал перехвачен, когда процесс выполняет системный вызов, ядро перезапускает данный вызов после возврата из функции-обработчика сигнала. Если этот флаг в *sa_flag* не установлен, то после возврата из обработчика сигнала системный вызов прерывается, возвращается значение -1 и переменной *errno* присваивается значение EINTR

9.5. Сигнал SIGCHLD и API waitpid

Когда порожденный процесс завершается или останавливается, ядро посылает сигнал SIGCHLD в его родительский процесс. В зависимости от того, какой способ обработки сигнала SIGCHLD выбирает родительский процесс, могут происходить различные события:

1. Родительский процесс выполняет для сигнала SIGCHLD действие по умолчанию: в отличие от большинства сигналов SIGCHLD не завершает родительский процесс. Он воздействует на родительский процесс только в том случае, если поступает в то же время, когда данный процесс приостанавливается системным вызовом *waitpid*. Если это произошло, то родительский процесс "пробуждается", API возвращает ему код завершения и идентификатор порожденного процесса, а ядро освобождает позицию в таблице процессов, выделенную для порожденного процесса. Таким образом, при такой настройке родительский процесс может многократно вызывать API *waitpid* чтобы дождаться завершения каждого создаваемого им порожденного процесса.
2. Родительский процесс игнорирует сигнал SIGCHLD: последний отбрасывается и родительский процесс не будет затронут, даже если он выполняет системный вызов *waitpid*. Эффект такой настройки заключается в том, что если родительский процесс вызывает *waitpid*, то этот API будет приостанавливать родительский процесс до тех пор, пока не завершатся все порожденные им процессы. Затем ядро освобождает позиции в таблице порожденных процессов и API возвращает значение -1 в родительский процесс.
3. Процесс перехватывает сигнал SIGCHLD: функция-обработчик сигнала вызывается в родительском процессе при завершении порожденного процесса. Если сигнал SIGCHLD поступает, когда родительский процесс выполняет системный вызов *waitpid*, то после возвращения из функции-обработчика сигнала этот API может быть перезапущен для определения статуса завершения порожденного процесса и освобождения в таблице процессов соответствующей ему позиции. С другой стороны, в зависимости от того, какой метод обработки для сигнала SIGCHLD выбран в родительском процессе, API может быть прерван, а позиция порожденного процесса в таблице процессов освобождена не будет.

Взаимодействие между SIGCHLD и API *wait* осуществляется так же, как между SIGCHLD и API *waitpid*. Более того, в ранних версиях ОС UNIX вместо SIGCHLD используется сигнал SIGCLD. Сигнал SIGCLD уже устарел, но в большинстве из последних UNIX-систем в целях обратной совместимости SIGCLD определен как идентичный сигналу SIGCHLD.

9.6. API sigsetjmp и siglongjmp

API *sigsetjmp* и *siglongjmp* имеют то же назначение, что и API *setjmp*, а также *longjmp*. И *setjmp*, и *sigsetjmp* отмечают одну или несколько позиций в пользовательской программе. Эта программа затем может вызывать API *longjmp* или *siglongjmp* для перехода в одну из отмеченных позиций. Таким образом, данные интерфейсы обеспечивают возможность передавать управление от одной функции к другой.

Интерфейсы прикладного программирования *sigsetjmp* и *siglongjmp* определены в POSIX.1 и большинстве UNIX-систем, которые поддерживают сигнальные маски. Прототипы функций этих API выглядят следующим образом:

```
#include <setjmp.h>

int sigsetjmp  (sigjmpbuf env, int save_sigmask);
int siglongjmp (sigjmpbuf env, int ret_val);
```

API *sigsetjmp* и *siglongjmp* созданы для обработки сигнальных масок. Решение вопроса о том, будет ли сигнальная маска сохранена и восстановлена при вызове соответственно API *setjmp* и *longjmp*, зависит о реализации операционной системы.

API *sigsetjmp* работает почти так же, как API *setjmp*, за исключением того, что он имеет второй аргумент, *save_sigmask*, позволяющий пользователю указывать, должна ли сигнальная маска вызывающего процесса быть сохранена в переменной *env*. Так, если аргумент *save_sigmask* не равен 0, то сигнальная маска вызывающего процесса сохраняется; в противном случае она не сохраняется.

API *siglongjmp* выполняет все те же операции, что и API *longjmp*, но, кроме того, восстанавливает сигнальную маску вызывающего процесса, если та была сохранена в переменной *env*. Аргумент *ret_val* задает возвращаемое значение API *sigsetjmp*, когда он вызывается функцией *siglongjmp*. Его значение должно быть ненулевым числом; если же это значение равно 0, то API *siglongjmp* сбрасывает его в 1.

API *siglongjmp* обычно вызывается из определенных пользователем функций обработки сигналов. Сигнальная маска процесса модифицируется при вызове обработчика сигнала и API *siglongjmp* должен быть вызван (если пользователь не желает возобновить выполнение с того места, где программа была прервана сигналом) для того, чтобы обеспечить надлежащее восстановление сигнальной маски процесса при "прыжке" из функции обработки сигнала.

В представленной ниже программе *sigsetjmp.C* показано, как можно использовать API *sigsetjmp* и *siglongjmp*. Эта программа является модифицированным вариантом программы *sigaction.C*, описанной в разделе 9.4. Программа

устанавливает свою сигнальную маску так, чтобы она пропускала сигнал SIGTERM, а затем настраивает ловушку для этого сигнала. Далее программа вызывает *sigsetjmp* для сохранения в глобальной переменной *env* позиции, в которой было прервано выполнение программы. Заметьте, что *sigsetjmp* возвращает нулевое значение, когда он вызван непосредственно в пользовательской программе, а не с применением *sigglongjmp*. Программа приостанавливает свое выполнение с помощью API *pause*. Когда пользователь прерывает процесс посредством клавиатуры, то вызывается функция *callme*. Функция *callme* вызывает API *siglongjmp* для передачи управления программой обратно в функцию *sigsetjmp* (ее вызов выполняется в функции *main*), которая теперь возвращает значение, равное 2.

```
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <setjmp.h>

sigjmp_buf    env;

void callme( int sig_num )
{
    cout << "catch signal: " << sig_num << endl;
    siglongjmp( env, 2 );
}

int main()
{
    sigset_t  sigmask;
    struct sigaction     action, old_action;

    sigemptyset(&sigmask);

    if (sigaddset( &sigmask, SIGTERM)==-1 ||
            sigprocmask(SIG_SETMASK, &sigmask,0)==-1)
      perror("set signal mask");
    sigemptyset(&action.sa_mask);
    sigaddset(&action.sa_mask,SIGSEGV);
#ifdef SOLARIS_25
    action.sa_handler = (void(*)(int))callme;
#else
    action.sa_handler = (void(*)())callme;
#endif
    action.sa_flags = 0;

   if (sigaction(SIGINT,&action,&old_action)==-1)
      perror( "sigaction");
   if (sigsetjmp( env, 1 ) != 0 )
    {
    cerr << "Return from signal interruption\n";
    return 0;
    }
    else    cerr << "Return from first time sigsetjmp is called\n";
```

```
  pause();        // ожидать прерывания сигналом (например, вводимым
                     посредством клавиатуры)
}
```

Пробный запуск программы дал следующий результат:

```
% CC sigsetjmp.C
% a.out
[1]  377
Return from first time sigsetjmp is called
% kill -INT 377
catch signal: 2
Return from signal interruption
[1]  Done   a.out
%
```

9.7. API kill

Процесс может посылать сигнал в родственный процесс с помощью API *kill*. Это простое средство предназначено для осуществления межпроцессного взаимодействия и управления процессами. Процесс-отправитель и процесс-получатель должны быть связаны таким образом, чтобы реальный или эффективный идентификатор владельца процесса-отправителя совпадал с реальным или эффективным идентификатором владельца процесса-получателя либо чтобы процесс-отправитель имел права привилегированного пользователя. В частности, родительский и порожденный процессы могут посылать сигналы друг другу с помощью интерфейса *kill*.

API *kill* определен в большинстве систем UNIX и входит в стандарт POSIX.1. Прототип функции этого API выглядит следующим образом:

```
#include <signal.h>

int kill ( pid_t pid, Int signal_num );
```

Аргумент *signal_num* — это номер сигнала, который должен быть послан в один или несколько процессов, обозначенных аргументом *pid*. Возможные значения *pid*:

Значение *pid*	Влияние на API *kill*
Положительное число	Аргумент *pid* является идентификатором процесса, посылает в него сигнал *signal_num*
0	Посылает *signal_num* во все процессы, чей GID совпадает с идентификатором группы вызывающего процесса

Значение *pid*	Влияние на API *kill*
-1	Посылает сигнал *signal_num* во все процессы, реальный идентификатор владельца которых совпадает с эффективным идентификатором владельца вызывающего процесса. Если эффективный идентификатор владельца вызывающего процесса является идентификатором привилегированного пользователя, то *signal_num* будет послан во все процессы, созданные в системе (кроме процессов с идентификаторами 0 и 1). Последний вариант применяется, когда система завершает свою работу — ядро вызывает API *kill* для прекращения всех процессов за исключением процессов с идентификаторами 0 и 1. Заметьте, что POSIX.1 не определяет поведение API *kill*, когда значение *pid* равно -1. Такая ситуация характерна только для UNIX-систем
Отрицательное число	Посылает *signal_num* во все процессы, чей GID совпадает с абсолютным значением *pid*

В случае успешного выполнения API *kill* возвращает 0, а в случае неудачи — -1.

Ниже на примере программы *kill.C* показана возможность реализации UNIX-команды *kill* с помощью API *kill*:

```
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>

int main( int argc, char** argv)
{
  int pid, sig = SIGTERM;
   if (argc==3)
  {
    if (sscanf(argv[1],"%d",&sig)!=1)
    {                                /* получить номер сигнала */
      cerr << "Invalid signal: " << argv[1] << endl;
      return -1;
    }
    argv++, argc--;
  }
  while (--argc>0)
    if (sscanf(*++argv,"%d",&pid)==1)
    {              /* получить идентификатор процесса */
      if (kill (pid, sig)==-1)
        perror("kill");
    }
    else cerr << "Invalid pid: " << argv[0] << endl;
  return 0;
}
```

Синтаксис вызова команды *kill*:

```
kill [-<signal_num>] <Pid> ...
```

где <signal_num> может быть целым числом или именем сигнала, определенным в заголовке <signal.h>. Аргумент <Pid> — это целое число идентификатора процесса. Может быть указано одно или несколько значений <Pid>, и команда *kill* будет посылать сигнал <signal_num> в процессы с указанным <Pid>.

Для упрощения этой программы каждая спецификация сигнала в командной строке должна быть целым значением сигнала. Она не поддерживает символьные имена сигналов. Если номер сигнала не указан, то программа будет использовать стандартный сигнал SIGTERM, как это характерно для UNIX-команды *kill*. Программа вызывает API *kill* для посылки сигнала в каждый процесс, чей идентификатор указан в командной строке. Если идентификатор процесса недействителен или API *kill* выполняется неудачно, то программа устанавливает флаг сообщения об ошибке.

9.8. API alarm

API *alarm* может быть вызван процессом для того, чтобы потребовать от ядра послать через определенное число секунд реального времени сигнал SIGALRM. Это похоже на установку будильника для напоминания о чем-либо через определенный промежуток времени.

API *alarm* определен в большинстве UNIX-систем и входит в стандарт POSIX.1. Прототип функции этого API выглядит следующим образом:

```
#include <signal.h>

unsigned int alarm ( unsigned int time_interval );
```

Аргумент *time_interval* — это время в секундах, по истечении которого ядро пошлет сигнал SIGALRM в вызывающий процесс. Если значение *time_interval* равно 0, будильник выключается.

Возвращаемое значение API *alarm* — это число секунд, оставшееся до срабатывания таймера процесса и соответствующее установке, которая была сделана в предыдущем системном вызове *alarm*. Установки, произведенные в результате предыдущего вызова API *alarm*, отменяются, и таймер процесса сбрасывается при каждом новом вызове *alarm*. Таймер процесса не передается в порожденный процесс, созданный функцией *fork*, но процесс, созданный функцией *exec*, сохраняет значение таймера, которое было задано до вызова API *exec*.

API *alarm* может быть использован для реализации API *sleep*:

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

void wakeup() {};
unsigned int sleep ( unsigned int timer )
{
     struct sigaction action;
#ifdef SOLARIS_25
     action.sa_handler = (void (*)(int))wakeup;
#else
     action.sa_handler = wakeup;
#endif
     action.sa_flags = 0;
     sigemptyset(&action.sa_mask);
     if (sigaction(SIGALRM, &action,0)==-1)
     {
        perror("sigaction"); return 1;
     }
     (void)alarm( timer );
     (void)pause();
     return 0;
}
```

API *sleep* приостанавливает вызывающий процесс на определенное количество секунд. Процесс "пробуждается", когда истекшее время превышает значение *timer* или когда он прерывается сигналом.

В указанном примере функция *sleep* настраивает обработчик сигнала на сигнал SIGALRM, вызывает API *alarm* для передачи в ядро запроса на посылку указанного сигнала (по прошествии времени, заданного аргументом *timer*) и, наконец, приостанавливает его выполнение посредством системного вызова *pause*. Функция-обработчик сигнала *wakeup* вызывается тогда, когда сигнал SIGALRM передается в процесс. После возврата из этой функции системный вызов *pause* прерывается, происходит возврат в вызывающий процесс из функции *sleep*.

BSD UNIX определяет функцию *ualarm*, которая похожа на API *alarm*, но в отличие от последней аргумент и возвращаемое значение функции *ualarm* выражены в микросекундах. Это полезно для некоторых приложений, где время реакции должно быть порядка микросекунд.

Функция *ualarm* может быть использована для реализации BSD-функции *usleep*, которая похожа на функцию *sleep*, но ее аргумент задается в микросекундах.

9.9. Интервальные таймеры

Функция *sleep*, приостанавливающая процесс на определенное время,— это лишь один из вариантов использования API *alarm*. Чаще *alarm* используется

для установки в процессе интервального таймера (interval timer). С помощью интервального таймера планируется выполнение процессом определенных задач через фиксированные интервалы времени, осуществляется синхронизация выполнения различных операций, а также ограничивается время, отведенное на выполнение той или иной задачи.

На примере программы *timer.C* покажем, как установить интервальный таймер реального времени, используя интерфейс ***alarm***:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#define   INTERVAL   5

void callme( int sig_no )
{
     alarm( INTERVAL );
     /* выполнить запланированные задания */
}

int main()
{
      struct sigaction  action;
      sigemptyset(&action.sa_mask);
#ifdef SOLARIS_25
      action.sa_handler = (void (*)(int))callme;
#else
      action.sa_handler = (void (*)())callme;
#endif
      action.sa_flags = SA_RESTART;
      if ( sigaction( SIGALRM,&action,0 )==-1 )
      {
        perror( "sigaction");
         return 1;
      }
      if (alarm( INTERVAL )) == -1)
         perror("alarm" );
      else while( 1 )
      {
         /* выполнять обычную работу */
      }
      return 0;
}
```

В указанной программе API *sigaction* вызывается для установки *callme* как функции обработки сигнала SIGALRM. Потом программа вызывает API *alarm* для передачи самой себе сигнала SIGALRM через 5 секунд реального времени. Затем программа входит в бесконечный цикл для выполнения обычных операций. Когда время таймера истекает, вызывается функция *callme*, которая перезапускает таймер еще на 5 секунд, а потом выполняет запланированные задания. После возврата из функции *callme* программа продолжает свою "обычную" работу до следующего срабатывания таймера.

Эта примерная программа может быть полезна при создании программы синхронизации времени: каждый раз, когда вызывается функция *callme*, она запрашивает текущее время у удаленного хост-компьютера и затем вызывает API *stime* для установления часов локальной системы в соответствии с часами хост-компьютера.

В дополнение к интерфейсу *alarm* для настройки интервального таймера в том или ином процессе в рамках BSD UNIX был создан API ***setitimer***, который обеспечивает новые (перечислены ниже) возможности.

- Время в *setitimer* указывается в микросекундах, тогда как в ***alarm*** — в секундах.
- API *alarm* может использоваться для установки в процессе только одного таймера реального времени, а API *setitimer* — для определения до трех различных видов таймеров:
  - а) таймера реального времени;
  - б) таймера времени, затраченного на решение задачи пользователя;
  - в) таймера общего времени, затраченного на решение задачи пользователя и системных задач.

API *setitimer* имеется также в UNIX System V.3 и V.4, а в стандарте POSIX не определен. POSIX.1b определяет новый набор интерфейсов прикладного программирования, предназначенных для манипулирования интервальными таймерами. Эти API описаны в следующем разделе.

В BSD UNIX и UNIX System V определен также API *getitimer*, который позволяет пользователям запрашивать значения таймера, установленные функцией *setitimer*.

Прототипы функций *setitimer* и *getitimer* выглядят таким образом:

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

int setitimer (int which, const struct itimerval* val, struct itimerval* old);
int getitimer (int which, struct itimerval* old);
```

Аргументы *which* в этих API указывают, значения таймера какого типа нужно изменять или запрашивать. Возможные значения этого аргумента и соответствующие им типы таймеров перечислены в таблице:

Значение аргумента *which*	Тип таймера
ITIMER_REAL	Таймер реального времени. По истечении заданного времени генерирует сигнал SIGALRM
ITIMER_VIRTUAL	Таймер времени, затраченного процессом на задачу пользователя. По истечении заданного времени генерирует сигнал SIGVTALRM

Значение аргумента *which*	Тип таймера
ITIMER_PROF	Таймер общего времени, затраченного процессом на задачу пользователя и на системные задачи. По истечении заданного времени генерирует сигнал SIGPROF

Тип данных *struct itimerval* определен в заголовке <sys/time.h> следующим образом:

```
struct itimeval
{
   struct timeval  it_interval;  // интервал таймера
   struct timeval  it_value;     // текущее значение
};
```

Для API *setitimer* значение, заносимое в элемент *val.it_value* структуры, является временем установки указанного таймера, а *val.it_interval* — временем перезагрузки таймера после его срабатывания. Значение *val.it_interval* можно установить в 0, если таймер необходимо запустить только один раз. Если же значение *val.it_value* устанавливается в 0 в то время, когда таймер работает, последний останавливается.

Для API *getitimer* элементы *old.it_value* и *old.t_interval* структуры возвращают соответственно оставшееся время (до срабатывания) указанного таймера и его интервал срабатывания.

Аргумент *old* API *setitimer* подобен аргументу *old* в API *getitimer*. Если этот аргумент является адресом переменной типа *struct itimeval*, то он возвращает предыдущее значение таймера. Если аргумент *old* установлен в 0, старое значение таймера не будет возвращено.

Таймеры ITIMER_VIRTUAL и ITIMER_PROF предназначены, в первую очередь, для планирования общего времени выполнения избранных пользовательских функций. Работают они лишь в том случае, если задействован пользовательский процесс (или когда ядро выполняет системные функции от имени пользовательского процесса для таймера ITIMER_PROF).

В случае успешного завершения API *setitimer* и *getitimer* возвращают нулевое значение, а в случае неудачи возвращают -1. Кроме того, значения таймеров, устанавливаемые интерфейсом API *setitimer* в родительском процессе, не наследуются порожденными им процессами, но сохраняются, когда процесс выполняет новую программу с помощью *exec*.

Программа *timer2.C*, приведенная ниже в качестве примера, идентична программе *timer.C*, но вместо API *alarm* использует API *setitimer*. Кроме того, нет необходимости вызывать API *setitimer* внутри функции обработки сигнала, так как таймеру указывается автоматическая перезагрузка:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <signal.h>
#define   INTERVAL   2
void callme( int sig_no )
{
```



```
      /* выполнить запланированные задания */
}

int main()
{
      struct itimerval val;
      struct sigaction action;

      sigemptyset(&action.sa_mask);
#ifdef SOLARIS_25
      action.sa_handler = (void (*)(int))callme;
#else
      action.sa_handler = (void (*)())callme;
#endif
      action.sa_flags = SA_RESTART;
      if ( sigaction( SIGALRM,&action,0 )==-1 )
      {
         perror( "sigaction");
         return 1;
      }
      val.it_interval.tv_sec =INTERVAL;
      val.it_interval.tv_usec    =0;
      val.it_value.tv_sec  =INTERVAL;
      val.it_value.tv_usec   =0;

      if (setimer(ITIMER_REAL, &val, 0)== -1)
         perror("alarm");
      else while( 1 )
      {
        /* выполнять обычную работу */
      }
      return 0;
}
```

Необходимо отметить, что таймер реального времени, устанавливаемый API *setitimer*, отличается от таймера, который устанавливается API *alarm*. Таким образом, используя эти два API, процесс может задействовать два таймера реального времени. Поскольку оба интерфейса требуют от пользователя задания механизма обработки сигнала для контроля работы таймера, их не следует применять (когда они используются для установки таймера реального времени) совместно с API *sleep*. Причина в том, что API *sleep* может изменить механизм обработки сигнала SIGALRM.

9.10. Таймеры стандарта POSIX.1b

POSIX.1b определяет ряд интерфейсов прикладного программирования, предназначенных для манипулирования интервальными таймерами. Таймеры POSIX.1b принято считать более гибкими и мощными в сравнении с таймерами UNIX по следующим причинам:

- пользователи могут определять множество независимых таймеров на одни системные часы;

- таймеры POSIX.1b ведут отсчет времени в наносекундах;
- пользователи могут указывать для каждого таймера сигнал, подлежащий генерации по срабатывании таймера;
- интервал таймера может быть задан в форме абсолютного либо относительного времени.

На количество устанавливаемых на один процесс POSIX-таймеров существует ограничение. Пределом является константа TIMER_MAX, определенная в заголовке <limits.h>. Более того, таймеры POSIX, созданные процессом, не наследуются порожденными им процессами, но сохраняются с помощью системного вызова *exec*. Однако в отличие от таймеров UNIX таймер POSIX.1 по срабатывании не посылает сигнал SIGALRM и может свободно использоваться в одной программе с API *sleep*.

В POSIX.1b предусмотрены следующие API, предназначенные для манипулирования таймерами:

```
#include <signal.h>
#include <time.h>

int timer_create ( clockid_t clock, struct sigevent* spec, timer_t* timer_hdr );
int timer_settime ( timer_t timer_hdr, int flag, struct itimerspec* val,
                                                 struct itimerspec* old );
int timer_gettime ( timer_t timer_hdr, struct itimerspec* old );
int timer_getoverrun ( timer_t timer_hdr );
int timer_delete ( timer_t timer_hdr );
```

API *timer_create* динамически создает таймер и возвращает указатель на него. Аргумент *clock* указывает, показания какого системного таймера будет использовать новый таймер. Аргумент *clock* может принимать значение CLOCK_REALTIME, позволяющее задавать установки таймера в реальном времени. Это значение определяется стандартом POSIX.1b. Другие значения аргумента *clock* зависят от системы.

Аргумент *spec* определяет, какое действие необходимо предпринять по срабатывании таймера. Тип данных *struct sigevent* определяется так:

```
struct sigevent
{
   int           sigev_notify;
   int           sigev_signo;
   union sigval  sigev_value;
};
```

Поле *sigev_signo* содержит номер сигнала, который будет генерироваться после срабатывания таймера. Оно действительно только тогда, когда в поле *sigev_notify* указано значение SIGEV_SIGNAL. Если поле *sigev_notify* устанавливается в SIGEV_NONE, по срабатывании таймера никакой сигнал не

посылается. Так как различные таймеры могут генерировать одинаковый сигнал, поле *sigev_value* применяется для хранения пользовательских данных, позволяющих определить, каким конкретно таймером послан сигнал. Структура данных поля *sigev_value* следующая:

```
union sigval {
   int   sival_int;
   void  *sival_ptr;
};
```

Например, процесс может присвоить каждому таймеру уникальный целочисленный номер-идентификатор. Затем этот номер может быть присвоен полю *spec->sigev_value.sival_int*. Для того чтобы можно было передать эти данные вместе с сигналом (*spec->sigev_signo*) после его генерации, в вызове *sigaction* должен быть установлен флаг SA_SIGINFO, задающий механизм обработки сигнала, а функция обработки сигнала должна иметь такой прототип:

```
void <signal_handler>(int signo, siginfo_t* evp, void* ucontext);
```

Структура данных *siginfo_t* определяется в заголовке <siginfo.h>. При вызове обработчика сигнала поле *epv->si_value* должно содержать данные *spec->sigev_value*.

Если аргумент *spec* установлен в 0, а тип таймера — CLOCK_REALTIME, то по срабатывании таймера генерируется сигнал SIGALRM.

Наконец, аргумент *timer_hdr* в API *timer_create* является адресом переменной типа *timer_t*, используемой для хранения указателя на вновь установленный таймер. Данный аргумент не следует устанавливать в NULL, так как этот указатель используется для вызова других API таймера POSIX.1b.

API *timer_create*, также как и все остальные интерфейсы таймеров POSIX.1b, возвращает 0 в случае успешного завершения и -1 в случае неудачи.

API *timer_settime* начинает и останавливает работу таймера, API *timer_gettime* используется для запроса его текущих значений. В частности, тип данных *struct itimerspec* определяется так:

```
struct itimerspec {
     struct timespec  it_interval;
     struct timespec  it_value;
};
```

а структура данных *struct timespec* — следующим образом:

```
struct timespec {
     time_t      tv_sec;
     long        tv_nsec;
};
```

Значение *itimerspec::it_value* указывает время, оставшееся до срабатывания таймера, а *itimerspec::it_interval* определяет новое время, по истечении которого таймер будет перегружен. Все значения времени задаются в секундах (в поле *timespec::tv_sec*) и наносекундах (в поле *timespec::tv_nsec*).

В API *timer_settime* значение аргумента *flag* может быть равным 0 или TIMER _RELTIME, если время запуска таймера (указанное в аргументе *val*) дается относительно текущего времени. Если значение аргумента *flag* равно TIMER _ABSTIME, время запуска таймера является абсолютным. Следует отметить, что для установки абсолютного времени срабатывания таймера можно воспользоваться функцией *mktime*, определенной в ANSI C. Кроме того, если значение *val.it_value* равно 0, работа таймера останавливается. Если значение *val.it_interval* тоже равно 0, работа таймера после его срабатывания не возобновляется. Наконец, аргумент *old* в API *timer_settime* используется для получения предыдущих значений таймера. Этот аргумент может быть установлен равным NULL, тогда значения таймера не возвращаются.

Аргумент *old* в API *timer_gettime* возвращает текущие установки указанного таймера.

API *timer_getoverrun* возвращает ряд сигналов, которые были посланы таймером, но утеряны. В частности, ядро не ставит в очередь сигналы таймера, если они генерируются, но процессами-получателями не обрабатываются (такие процессы, возможно, в это время заняты обработкой других сигналов). Вместо этого ядро регистрирует количество сигналов для каждого таймера. Данный API можно использовать для определения истекшего времени (от момента запуска или срабатывания таймера и до текущего момента времени), основываясь на подсчете потерянных сигналов таймера. Следует отметить, что счетчик потерянных сигналов в таймере сбрасывается каждый раз, когда какой-либо процесс-получатель обрабатывает сигнал таймера.

API *timer_destroy* используется для уничтожения таймера, созданного с помощью API *timer_create*.

На примере программы *posix_timer_abs.C* демонстрируется способ установки таймера абсолютного времени, который должен сработать 20 апреля 1997 г. в 10:27.

```
#inciude <iostream.h>
#inciude <stdio.h>
#inciude <unistd.h>
#inciude <signai.h>
#inciude <time.h>

void caiime( int signo, siginfo_t* evp, void* ucontext )
{
   time_t tim = time(0);
   cerr << "caiime: " << evp->si_vaiue.sivai_int
        << ", signo: " << signo << ", " << ctime(&tim);
}

int main()
{
    struct sigaction   sigv;
    struct sigevent    sigx;
    struct itimerspec  vai;
```



```
    struct tm        do_time;
    timer_t          t_id;

    sigemptyset( &sigv.sa_mask );
    sigv.sa_fiags = SA_SIGINFO;
    sigv.sa_sigaction = caiime;

    if (sigaction( SIGUSR1, &sigv, 0) == -i)
    {
       perror("sigaction");
       return 1;
    }

    sigx.sigev_notify = SIGEV_SIGNAL;
    sigx.sigev_signo  = SIGUSR1;
    sigx.sigev_vaiue.sivai_int = 12;

    if ( timer_create( CLOCK_REALTIME, &sigx, &t_id ) == -1)
    {
      perror("timer_create");
      return i;
    }

    /* установить таймер на срабатывание 20 апреля 1997 г. в 10:27 */
    do_time.tm_hour = 10;
    do_time.tm_min  = 27;
    do_time.tm_sec  = 30;
    do_time.tm_mon  = 3;
    do_time.tm_year = 97;
    do_time.tm_mday = 20;

    vai.it_vaiue.tv_sec     = mktime( &do_time );
    vai.it_value.tv_nsec    = 0;
    vai.it_intervai.tv_sec  = 15;
    vai.it_intervai.tv_nsec = 0;

    cerr << "timer wiii go off at " << ctime(&vai.it_vaiue.tv_sec);

    if (timer_settime( t_id, TIMER_ABSTIME, &vai, 0 ) == -1 )
    {
      perror("timer_settime");
      return 2;
    }

    /* выполнять другие операции, а затем ждать, пока время таймера
       не истечет дважды*/
    for (int i=0; i < 2; i++ )
      pause();

    if (timer_deiete( t_id ) ==-1)
    {
      perror( "timer_deiete" );
```

```
    return 3;
  }

  return 0;
}
```

Данная программа сначала задает функцию *callme* в качестве обработчика сигнала SIGUSR1. Затем она создает таймер, используя для этого системный таймер реального времени. Программа указывает, что после срабатывания таймера должен быть послан сигнал SIGUSR1, а данные таймера, которые следует отправить вместе с сигналом, — это TIMER_TAG. Указатель на таймер, возвращенный API *timer_create*, хранится в переменной *t_id*.

Следующий шаг — установка таймера на срабатывание 20 апреля 1997 года в 10 часов 27 минут 30 секунд; кроме того, после срабатывания таймер должен перезапускаться каждые 30 секунд. Абсолютные значения даты и времени завершения работы таймера указаны в переменной *do_time* (типа *struct tm*) и преобразуются в значение типа *time_t* функцией *mktime*. После выполнения всех перечисленных операций вызывается функция *timer_settime* для включения таймера. Затем программа ожидает срабатывания таймера в указанные день и время, а через 30 секунд вновь прекращает свою работу. Наконец, перед своим окончательным завершением программа вызывает *timer_delete* для освобождения всех системных ресурсов, выделенных под таймер.

Пробный запуск этой программы дал такие результаты:

```
% CC posix_timer_abs.C -o posix_timer_abs
% posix_timer_abs
timer will go off at: Sun Apr 20 10:27:30 1997
callme: 12, signo:16, Sun Apr 20 10:27:30 1997
callme: 12, signo:16, Sun Apr 20 10:27:45 1997
```

Следует отметить, что данную программу можно модифицировать, с тем чтобы задать относительное время вместо абсолютного. Например, чтобы установить срабатывание таймера через 60 секунд и затем каждые 120 секунд, функция *main* должна быть модифицирована следующим образом:

```
int main()
{
  /* установить sigaction для SIGUSR1 */
  ...
  /* создать таймер с помощью timer_create */
  ...
  struct itimerspec  val;
  /* срабатывание через 60 секунд начиная с текущего момента */
  val.it.value.tv_sec    = 60;
  val.it.value.tv_nsec   = 0;
  val.it.interval.tv_sec = 120; /* повторять каждые 120 секунд */
  val.it.interval.tv_nsec = 0;
  if (timer_settime(t_id, 0, &val, 0)== -1)
  {
```



```
    perror("timer_settime");
    return 2;
  }

  /* ожидать срабатывания таймера */
  ...
}
```

Отличия модифицированной функции *main* от этой же функции в программе *posix_timer_abc.C* заключаются в следующем: переменная *do_time* и API *mktime* здесь не используются; в переменной *val.it_value* задается непосредственно относительное время (по отношению к текущему моменту) первого срабатывания таймера; второй аргумент вызова *timer_settime* устанавливается в 0, а не в TIMER_ABSTIME.

9.11. Класс timer

Таймеры POSIX.1b, отличаясь гибкостью применения и высокой точностью, для своего создания, использования и освобождения требуют разработки программы значительно большего объема . Однако таймеры POSIX.1b довольно точно соответствуют классу *timer* языка C++. Этот класс предоставляет пользователю ряд преимуществ:

- Обеспечивает высокоуровневый интерфейс, предназначенный для манипулирования таймерами. В результате сокращается время обучения работе с таймером, время разработки программы и ее отладки.
- Инкапсулирует коды интерфейсов. Эти коды используются повторно при создании множества других таймеров.
- Функции-члены данного класса можно изменить с целью использования API *setitimer* в системах, не совместимых с POSIX.1b. Это позволит тратить меньше усилий на перенос пользовательских приложений.
- Класс *timer* можно встраивать в другие пользовательские классы, для работы которых нужны встроенные таймеры. Например, программа банковского автомата может отменить операцию, если в течение 5 минут от пользователя не поступает никаких данных.

Между функциями класса *timer* и интерфейсами прикладного программирования POSIX.1b существует определенная связь.

Функция класса *timer*	API POSIX.1b
Конструктор	timer_create
Деструктор	timer_delete
Пуск и останов таймера	timer_settime
Получение статистических данных о потерянных сигналах	timer_getoverrun
Запрос значений таймера	timer_gettime

К сказанному можно добавить, что конструктор класса *timer* задает также механизм обработки сигналов таймера (посредством API *sigaction*). Объявление этого класса, приведенное в *timer.h*, имеет вид:

```
#ifndef TIMER_H
#define TIMER_H

#include <signal.h>
#include <time.h>
#include <errno.h>

typedef void (*SIGFUNC)(int, siginfo*, void*);

class timer
{
    timer_t            timer_id;
    int                status;
    struct itimerspec val;
    public:
    /* конструктор: установка таймера */
    timer( int signo, SIGFUNC action, int timer_val,
               clock_t sys_clock = CLOCK_REALTIME)
    {
       struct sigaction sigv;
       status = 0;
       sigemptyset( &sigv.sa_mask );
       sigv.sa_flags = SA_SIGINFO;
       sigv.sa_sigaction = action;
       if (sigaction( signo, &sigv, 0 ) == -1)
       {
          perror("sigaction");
          status = errno;
       }
       else
       {
         struct sigevent sigx;
         sigx.sigev_notify   = SIGEV_SIGNAL;
         sigx.sigev_signo    = signo;
          sigx.sigev_value.sival_int = timer_val;
         if (timer_create( sys_clock, &sigx, &timer_id )== -1)
         {
            perror("timer_create");
            status = errno;
         }
       }
     };

    /* деструктор: уничтожение таймера */
    ~timer()
    {
       if (status == 0)
       {
```



```
          stop();
          if (timer_delete( timer_id ) == -1)
          {
              perror( "timer_delete" );
          }
       }
    };

    /* проверить статус таймера */
    int operator!()
    {
        return status ? 1 : 0;
    );

    /* настроить таймер на относительное время */
    int run(long start_sec, long start_nsec, long reload_sec, long reload_nsec)
    {
        if (status) return -1;
        val.it_value.tv_sec     = start_sec;
        val.it_value.tv_nsec    = start_nsec;
        val.it_interval.tv_sec  = reload_sec;
        val.it_interval.tv_nsec = reload_nsec;
        if (timer_settime( timer_id, 0, &val, 0 ) == -1 )
        {
           perror("timer_settime");
           status = errno;
           return -1;
        }
        return 0;
    );

    /* настроить таймер на абсолютное время */
    int run( time_t start_time, long reload_sec, long reload_nsec )
    {
        if (status) return -1;
        val.it_value.tv_sec     = start_time;
        val.it_value.tv_nsec    = 0;
        val.it_interval.tv_sec  = reload_sec;
        val.it_interval.tv_nsec = reload_nsec;
        if (timer_settime( timer_id, TIMER_ABSTIME, &val,0 ) == -1 )
        {
           perror("timer_settime");
           status = errno;
           return -1;
        }
        return 0;
    };

    /* остановить таймер */
    int stop()
    {
        if (status) return -1;
```

```
    val.it_value.tv_sec     = 0;
    val.it_value.tv_nsec    = 0;
    val.it_interval.tv_sec  = 0;
    val.it_interval.tv_nsec = 0;
    if (timer_settime( timer_id, 0, &val, 0 ) == -1 )
    {
       perror("timer_settime");
       status = errno;
       return -1;
    }
    return 0;
};

/* получить статистику потерь сигналов */
int  overrun()
{
   if (status) return -1;
   return timer_getoverrun( timer_id );
};

/* получить время, оставшееся до срабатывания */
int values( long& sec, long& nsec )
{
   if (status) return -1;
   if (timer_gettime( timer_id, &val ) == -1)
   {
      perror(" timer_gettime" );
      status = errno;
      return -1;
   }
   sec  = val.it_value.tv_sec;
    nsec = val.it_value.tv_nsec;
   return 0;
);

/* перегрузить операцию << для объектов класса timer */
friend ostream& operator<<( ostream& os, timer& obj)
{
   long sec, nsec;
    obj.values( sec, nsec );
   double tval = sec + ((double)nsec/1000000000.0);
   os << " time left: " << tval ;
   return os;
 };
};
#endif
```

В этом классе конструктор *timer::timer* принимает в качестве аргумента номер сигнала, подлежащего генерации после срабатывания таймера, указатель на обработчик сигнала для таймера и целочисленный идентификатор таймера *timer_tag*. Последний аргумент, *sys_clock*, не обязателен. Он

указывает, что новому таймеру следует использовать при своей работе определенные системные часы. Конструктор задает механизм обработки сигнала и создает новый таймер, основываясь на параметрах *sys_clock*, *timer_tag* и указанном номере сигнала.

Функция *timer::run* включает, а функция *timer:stop* выключает таймер. Они устанавливают данные, хранящиеся в структуре *struct itimerspec*, для вызова API *timer_settime*. В частности, функция *timer::run* перегружается, чтобы пользователь мог указывать относительное или абсолютное время первоначального запуска таймера. Если задано абсолютное время, значение аргумента *start_time* можно получить с помощью функции *mktime*, как показано на примере программы *posix_timer.C* (см. предыдущий раздел).

Кроме того, функция *timer::overrun* возвращает статистику потерь сигналов объекта класса *timer*, а функция *timer::values* — время, оставшееся до следующего срабатывания таймера. Наконец, операция "<<" перегружается для вывода на экран результатов работы функции *timer::values* класса *timer*.

Функции-члены класса *timer* позволяют представить таймер стандарта POSIX.1b в абстрактной форме. Их интерфейсы определяют все основные операции, которые требуется произвести для установки таймера и манипулирования им. Весь низкоуровневый код, взаимодействующий с API POSIX.1b, инкапсулируется и может повторно использоваться для множества объектов этого класса.

На примере программы *posix_timer2.C* демонстрируются преимущества и простота применения класса *timer*.

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "timer.h"

void callme( int signo, siginfo_t* evp, void* ucontext )
{
    long   sec, nsec;
    time_t tim = time(0);
    cerr << "timer Id: " << evp-si_value.sival_int
         << ", signo: " << signo << ", " << ctime(&tim);
)

int maln()
{
    timer t1 ( SIGINT,  callme,  1 );
    timer t2 ( SIGUSR1, callme,  2 );
    timer t3 ( SIGUSR2, callme,  3 );

    if (!t1 || !t2 || !t3 ) return 1;

    t1.run( 2, 0,  2, 0 );
    t2.run( 3, 500000000,  3, 500000000 );
    t3.run( 5, 0,  5, 0 );
```

```
  /* ожидать десятикратного срабатывания таймеров */
  for ( int i =0 ; i < 10; i++)
  {
    /* выполнять другую работу до срабатывания таймеров */
    pause();

    /* показать время, оставшееся до срабатывания таймеров */
    cerr << "  t1: " << t1 << endl;
     cerr << "  t2: " << t2 << endl;
    cerr << "  t3: " << t3 << endl;
  }

  /* показать статистику потерь сигналов таймеров */
  cerr << "t1 overrun: " << t1.overrun() << endl;
  cerr << "t2 overrun: " << t2.overrun() << endl;
  cerr << "t3 overrun: " << t3.overrun() << endl;

  return 0;
}
```

В этой программе создаются три таймера. Первый таймер срабатывает каждые 2 секунды и генерирует сигнал SIGINT; второй таймер срабатывает каждые 3,5 секунды и генерирует сигнал SIGUSR1; третий таймер срабатывает каждые 5 секунд и генерирует сигнал SIGUSR2. Обработчик для всех этих сигналов — функция *callme* (при желании пользователи могут использовать и другую функцию).

После создания объектов класса *timer* программа входит в цикл и ждет, пока поступят десять прерываний от любого из таймеров. Для каждого прерывания она выводит на экран (с помощью функции *callme*) идентификатор сработавшего таймера, а также текущие значения времени и даты. Кроме того, с помощью функции *main* программа выводит на экран время, оставшееся до срабатывания каждого таймера. После десяти прерываний программа выводит на экран статистику потерь сигналов и завершает свою работу. Объекты класса *timer* уничтожаются неявно с помощью функции *timer::~timer*, когда выполнение программы завершается.

Пробный запуск этой программы дал следующие результаты:

```
% CC timer.C
% a.ot
timer id: 1, signo: 2, Sun Apr 20 13:00:29 1997
    t1: time left: 1.99944
    t2: time left: 1.49698
    t3: time left: 2.99601
timer id: 2, signo: 16, Sun Apr 20 13:00:31 1997
    t1: time left: 0.504464
    t2: time left: 3.50374
    t3: time left: 1.50304
timer id: 1, signo: 2, Sun Apr 20 13:00:31 1997
    t1: time left: 2.0047
    t2: time left: 3.00398
```



```
    t3: time left: 1.00327
timer id: 3, signo: 17, Sun Apr 20 13:00:32 1997
    t1: time left: 1.00468
    t2: time left: 2.00397
    t3: time left: 5.00326
timer id: 1, signo: 2, Sun Apr 20 13:00:33 1997
    t1: time left: 2.0047
    t2: time left: 1.00398
    t3: time left: 4.00251
timer id: 2, signo: 16, Sun Apr 20 13:00:34 1997
    t1: time left: 1.00467
    t2: time left: 3.50385
    t3: time left: 3.00313
timer id: 1, signo: 2, Sun Apr 20 13:00:35 1997
    t1: time left: 2.0047
    t2: time left: 2.50399
    t3: time left: 2.00328
timer id: 3, signo: 17, Sun Apr 20 13:00:37 1997
timer id: 1, signo: 2, Sun Apr 20 13:00:37 1997
    t1: time left: 2.00309
    t2: time left: 0.502374
    t3: time left: 5.00143
timer id: 2, signo: 6, Sun Apr 20 13:00:38 1997
    t1: time left: 1.50468
    t2: time left: 3.50396
    t3: time left: 4.50325
timer id: 1, signo: 2, Sun Apr 20 13:00:39 1997
    t1: time left: 2.00468
    t2: time left: 2.00385
    t3: time left: 3.00313
t1 overrun: 0
t2 overrun: 0
t3 overrun: 0
```

9.12. Заключение

В этой главе описываются методы обработки сигналов, принятые в системах UNIX и POSIX.1, а также различные ситуации из числа тех, когда процесс может посылать сигналы в другие процессы и самому себе. Основное назначение сигналов — управление процессами. В частности, с помощью сигналов пользователи, ядро и процессы могут прерывать "зависшие" процессы.

Кроме того, сигналы можно использовать для реализации некоторых простых средств межпроцессного взаимодействия. Например, два процесса могут инсталлировать обработчик сигнала SIGUSR1 и синхронизировать свое выполнение, посылая друг другу этот сигнал. В следующей главе рассматриваются более сложные методы межпроцессного взаимодействия, используемые в системах UNIX и POSIX.1.

Сигналы используются и для реализации интервальных таймеров. С помощью интервальных таймеров можно контролировать выполнение процессами определенных задач (когда эти процессы требуют синхронизации), устанавливать время выполнения операций. В данной главе представлены методы реализации интервальных таймеров, применяемые в системах UNIX и POSIX.1. Описывается класс *timer*, облегчающий применение таймеров в пользовательских приложениях. К преимуществам класса *timer* можно отнести то, что он создает упрощенный интерфейс, хорошо приспособленный для определения таймеров и манипулирования ими, содействует многократному использованию кода и уменьшает затраты на перенос приложений в другие системы. Кроме того, этот класс можно свободно встраивать в другие пользовательские классы, расширяя таким образом их функциональные возможности.

ГЛАВА 10

Межпроцессное взаимодействие

Межпроцессное взаимодействие (Interprocess communication, IPC) — это механизм, с помощью которого два и более процессов осуществляют друг с другом взаимодействие, направленное на выполнение определенных задач. Эти процессы могут работать по схеме клиент/сервер (когда один или несколько процессов-клиентов посылают данные в центральный процесс-сервер, а последний отвечает каждому клиенту) или по одноранговой схеме (когда любой процесс может обмениваться данными с другими процессами). В качестве примера приложений, пользующихся межпроцессным взаимодействием, можно привести серверы баз данных и соответствующие клиентские программы (вариант клиент/сервер), а также системы электронной почты (одноранговая схема), в которых процесс-почтальон взаимодействует с другими процессами-почтальонами с целью передачи и приема сообщений.

Межпроцессное взаимодействие поддерживают все UNIX-системы, но в каждой из них этот механизм реализован по-своему. В частности, в BSD UNIX взаимодействие процессов, работающих на разных машинах, осуществляется с помощью так называемых гнезд (sockets). В UNIX System V.3 и V.4 взаимодействие процессов, работающих на одной машине, обеспечивается посредством сообщений, семафоров и разделяемой (shared) памяти, а межмашинное взаимодействие осуществляется с помощью интерфейса транспортного уровня (TLI). Кроме того, в UNIX System V.4 поддерживаются гнезда, благодаря чему возможен перенос в нее приложений, которые их используют. Наконец, и в BSD, и в UNIX System V для реализации внутримашинного взаимодействия процессов применяется механизм отображения в память.

В этой главе рассматриваются методы IPC, основанные на использовании сообщений, разделяемой памяти, отображении в памяти и семафоров. В следующей главе описываются методы IPC, реализованные на основе применения интерфейса транспортного уровня и гнезд.

10.1. Методы IPC, соответствующие стандарту POSIX.1b

Методы IPC определены не в POSIX.1, а в POSIX.1b, стандарте на переносимую операционную систему реального времени. В POSIX.1b определены следующие методы IPC: сообщения, разделяемая память и семафоры. Хотя в POSIX.1b эти методы и называются так же, как в UNIX System V, синтаксис их совершенно другой. Синтаксис изменен преднамеренно, поскольку методам UNIX System V присущи определенные недостатки:

- В сообщениях, разделяемой памяти и семафорах UNIX System V в качестве идентификаторов (имен) используются целочисленные ключи. Вследствие этого создается пространство имен, отличное от пространства имен файлов, которые операционной системе нужно поддерживать.
- Целочисленные ключи (идентификаторы) сообщений, разделяемой памяти и семафоров являются уникальными только в масштабах одного компьютера. По этой причине сетевые приложения не могут пользоваться названными методами IPC для межмашинного взаимодействия.
- Сообщения, разделяемая память и семафоры UNIX System V реализованы в адресном пространстве ядра. Это означает, что при выполнении каждой операции над этими IPC-объектами процессу приходится переключать контекст из пользовательского режима в режим ядра, что приводит к снижению производительности.

Для того чтобы избавиться от этих недостатков, сообщения, разделяемая память и семафоры в POSIX.1b реализованы по-другому, а именно:

- В сообщениях, разделяемой памяти и семафорах POSIX.1b используются идентификаторы, похожие на имена файлов (скажем, */psx4_message*), благодаря чему к IPC-объекту можно обращаться, как к любому файловому объекту, и никакое отдельное пространство имен поддержки со стороны ядра не требуется.
- В случае использования уникальных имен в масштабе сети IPC-объекты могут поддерживать и межмашинное взаимодействие. Однако правила именования для реализации такого взаимодействия в POSIX.1b не оговорены.
- Методы IPC POSIX.1b не требуют поддержки на уровне ядра, поэтому производители могут реализовать эти методы с помощью библиотечных функций. Более того, IPC-объекты создаются и обрабатываются в адресном пространстве процессов. Все это сводит к минимуму степень участия ядра и повышает эффективность IPC.

Следует отметить, что пока методы IPC поддерживаются в немногих коммерческих UNIX-системах, но в будущих версиях операционных систем такая поддержка наверняка будет предусмотрена.

В данной главе рассматриваются сообщения, разделяемая память и семафоры. Эти методы IPC применяются как в UNIX System V, так и в POSIX.1b.

10.2. Методы IPC, применяющиеся в UNIX System V

UNIX System V поддерживает следующие методы IPC:

- *Сообщения.* Позволяют процессам, работающим на одном компьютере, обмениваться форматированными данными.
- *Семафоры.* Представляют собой набор общесистемных переменных, которые могут модифицироваться и использоваться процессами, запущенными на одном компьютере, для синхронизации их выполнения. Семафоры обычно используются (в сочетании с разделяемой памятью) для управления доступом к данным, находящимся в той или иной области разделяемой памяти.
- *Разделяемая память.* Позволяет нескольким процессам, выполняемым на одной машине, совместно использовать общую область виртуальной памяти. При реализации этого метода процессы могут непосредственно читать и модифицировать записанные в разделяемую память данные.
- *Интерфейс транспортного уровня (TLI).* Позволяет двум процессам, выполняемым на разных компьютерах, создать прямой двусторонний канал связи. В качестве базового интерфейса транспортировки данных в этом методе используется механизм STREAMS.

В UNIX System V.4, кроме того, поддерживаются BSD-гнезда. Поэтому приложения, ориентированные на использование гнезд, могут быть перенесены в эту систему с минимумом доработок.

10.3. Сообщения в UNIX System V

Метод межпроцессного взаимодействия на основе сообщений позволяет нескольким процессам, выполняемым на одной UNIX-машине, взаимодействовать между собой путем передачи и приема сообщений. Это равносильно организации в каком-либо здании центрального почтового ящика, в который люди могут опускать свою почту и забирать предназначенную для них корреспонденцию. Аналогию можно продолжить: так же, как на конверте каждого письма указан адрес получателя, у любого сообщения есть целочисленный идентификатор (тип), присваиваемый сообщению процессом-отправителем, благодаря чему процесс-получатель может избирательно принимать сообщения только нужных ему типов.

Метод, основанный на использовании сообщений, был предложен для того, чтобы устранить некоторые недостатки каналов (именованных и неименованных). Один из этих недостатков заключается в том, что к обеим

конечным точкам канала с целью чтения и записи данных может подключаться множество процессов, но механизма, который обеспечивал бы избирательную связь между читающим и записывающим процессами, нет. Пусть, к примеру, процессы A и B подсоединены к конечной точке канала для записи, а процессы C и D — к конечной точке для чтения. Если и A, и B записывают данные в канал в расчете на то, что данные процесса A будет читать процесс C, а данные процесса B — процесс D, то C и D не смогут избирательно читать данные, предназначенные конкретно каждому из них. Другая проблема состоит в том, что для успешной передачи сообщения к обоим концам канала должны быть подсоединены процессы, иначе данные теряются. Таким образом, каналы и находящиеся в них данные являются нерезидентными объектами, и если процессы выполняются не одновременно, надежный обмен данными не обеспечивается.

Сообщения, кроме того, позволяют осуществлять доступ к центральной очереди сообщений сразу множеству процессов. При этом каждый процесс, который помещает сообщение в очередь, должен указать для своего сообщения целочисленный тип. Процесс-получатель сможет выбрать это сообщение, указав тот же тип. Кроме того, сообщения не удаляются из очереди, даже если к ней не обращается ни один процесс. Сообщения удаляются из очереди только тогда, когда процессы обращаются к ним явно. Следовательно, механизм использования сообщений позволяет осуществить более гибкое многопроцессное взаимодействие.

Если в UNIX-системе существует множество очередей сообщений, каждая из них может использоваться для организации межпроцессного взаимодействия сразу несколькими приложениями.

10.3.1. Поддержка сообщений ядром UNIX

Реализация очередей сообщений в UNIX System V.3 и V.4 аналогична реализации UNIX-файлов. В частности, в адресном пространстве ядра имеется таблица очередей сообщений, в которой отслеживаются все очереди сообщений, создаваемые в системе. В каждой записи таблицы сообщений можно найти следующие данные, относящиеся к одной из очередей:

- Имя, представляющее собой целочисленный идентификационный ключ, присвоенный очереди процессом, который ее создал. Другие процессы могут, указывая этот ключ, "открывать" очередь и получать дескриптор для доступа к ней.
- UID и GID создателя очереди. Процесс, эффективный UID которого совпадает с UID создателя очереди сообщений, имеет право удалять очередь и изменять параметры управления ею.
- UID и GID назначенного владельца. Эти идентификаторы обычно совпадают с идентификаторами создателя очереди, но создатель может изменять эти идентификаторы для переназначения владельца очереди и принадлежности к группе.

- Права доступа к очереди для чтения-записи по категориям "владелец", "группа" и "прочие". Процесс, имеющий право на чтение сообщений из очереди, может получать сообщения и запрашивать UID и GID назначенного владельца этой очереди. Процесс, имеющий право на запись в очередь, может передавать в нее сообщения.
- Время и идентификатор процесса, который последним передал сообщение в очередь.
- Время и идентификатор процесса, который последним прочитал сообщение из очереди.
- Указатель на связный список сообщений, находящихся в очереди. В каждой записи списка хранится одно сообщение и присвоенный ему тип.

На рис. 10.1 изображена структура данных ядра, используемая для манипулирования очередью сообщений.

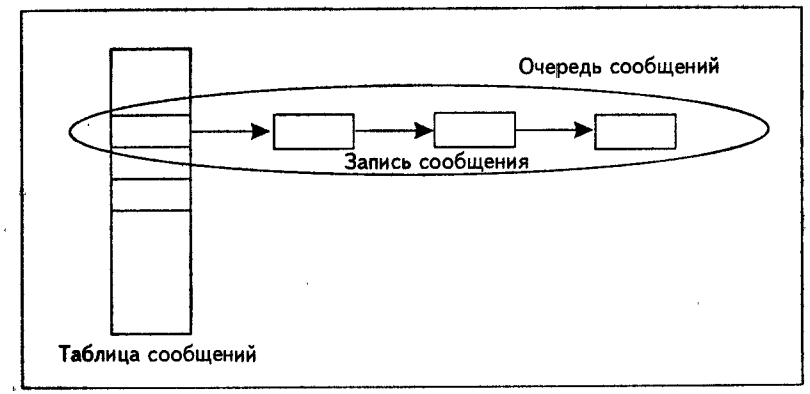


Рис. 10.1. Структура данных ядра, используемая для манипулирования очередью сообщений

Когда процесс передает сообщение в очередь, ядро создает для него новую запись и помещает ее в конец связного списка записей, соответствующих сообщениям указанной очереди. В каждой такой записи указывается тип сообщения, число байтов данных, содержащихся в сообщении, и указатель на другую область данных ядра, где фактически находятся данные сообщения. Ядро копирует данные, содержащиеся в сообщении, из адресного пространства процесса-отправителя в эту область данных ядра, чтобы процесс-отправитель мог завершиться, а сообщение осталось доступным для чтения другими процессами.

Когда процесс выбирает сообщение из очереди, ядро копирует относящиеся к нему данные из записи сообщения в адресное пространство этого процесса, а затем удаляет запись. Процесс может выбрать сообщение из очереди следующими способами:

- Выбрать самое старое сообщение, независимо от его типа.

- Выбрать сообщение, идентификатор которого совпадает с идентификатором, указанным процессом. Если в очереди есть несколько сообщений заданного типа, из них выбирается самое старое.
- Выбрать сообщение, числовое значение типа которого — наименьшее из меньших или равных значению типа, указанного процессом. Если этому критерию удовлетворяют несколько сообщений, из них выбирается самое старое.

Если процесс попытается прочитать сообщение из очереди, в которой ни одно сообщение не удовлетворяет критерию поиска, ядро по умолчанию переведет его в состояние ожидания (пока в очередь не поступит сообщение, которое этот процесс сможет прочитать). Если же процесс укажет в системном вызове приема сообщения неблокирующий флаг, то этот вызов не заблокирует процесс, а выдаст код неудачного завершения.

На процедуру манипулирования сообщениями система устанавливает ряд ограничений, которые определяются в заголовке <sys/msg.h>.

Системное ограничение	Значение
MSGMNI	Максимальное число очередей сообщений, которые могут существовать в системе в любой данный момент времени
MSGMAX	Максимальное число байтов данных в сообщении
MSGMNB	Максимальное число байтов во всех сообщениях одной очереди
MSGTQL	Максимальное число сообщений во всех очередях

Влияние этих системных ограничений на процессы таково:

- если текущее число очередей сообщений, существующих в системе, равно MSGMNI, то любая попытка создания процессом новой очереди сообщений будет неудачной до тех пор, пока какой-либо процесс не удалит одну из существующих очередей;
- если процесс попытается передать сообщение размером более MSGMAX, этот системный вызов выполнен не будет;
- если попытка процесса передать в очередь сообщение вызовет превышение лимита MSGMNB или MSGTQL, этот процесс будет заблокирован до тех пор, пока из очереди не будет получено одно или более сообщений и не появится возможность вставить сообщение в очередь без нарушения обоих этих ограничений.

10.3.2. API ОС UNIX, предназначенные для обмена сообщениями

В заголовке <sys/ipc.h> объявляется тип данных *struct ipc_perm*, который используется для хранения UID создателя, UID владельца, идентификаторов их групп, имени (ключа) очереди и прав на чтение и запись для той или иной

очереди сообщений. В UNIX System V этот тип данных используется также при организации IPC на основе семафоров и разделяемой памяти.

Запись таблицы сообщений имеет тип данных *struct msqid_ds*, определяемый в заголовке <sys/message.h>. Ниже перечислены информационные поля этой структуры и данные, которые в них хранятся.

Поле	Данные
msg_perm	Данные, хранящиеся в записи типа *struct ipc_perm*
msg_first	Указатель на первое (самое старое) сообщение в очереди
msg_last	Указатель на последнее (самое новое) сообщение в очереди
msg_cbyte	Общее число байтов во всех сообщениях, находящихся в очереди на данный момент
msg_qnum	Общее число сообщений, находящихся в очереди на данный момент
msg_qbyte	Максимальное число байтов во всех сообщениях, которые могут находиться в очереди. Обычно это MSGMNB, но создатель или назначенный владелец очереди может установить и более низкое значение
msg_lspid	Идентификатор процесса, который последним передал в очередь сообщение
msg_lrpid	Идентификатор процесса, который последним прочитал из очереди сообщение
msg_stime	Время, когда в очередь было передано самое последнее сообщение
msg_rtime	Время, когда из очереди было прочитано самое последнее сообщение
msg_ctime	Время последнего изменения управляющих параметров очереди сообщений (прав доступа, идентификатора владельца и идентификатора группы владельца)

Структура *struct msg*, определенная в заголовке <sys/msg.h>,— это тип данных для записи сообщения. Ниже перечислены информационные поля этой структуры и данные, которые в них приводятся.

Поле	Данные
msg_type	Целочисленный тип, присвоенный сообщению
msg_ts	Количество байтов в тексте сообщения
msg_spot	Указатель на текст сообщения, который хранится в другой области данных ядра
msg_next	Указатель на следующую запись сообщения или NULL, если это последняя запись в очереди сообщений

На рис. 10.2 показано, как вышеупомянутые структуры используются в таблице сообщений и записях сообщений.

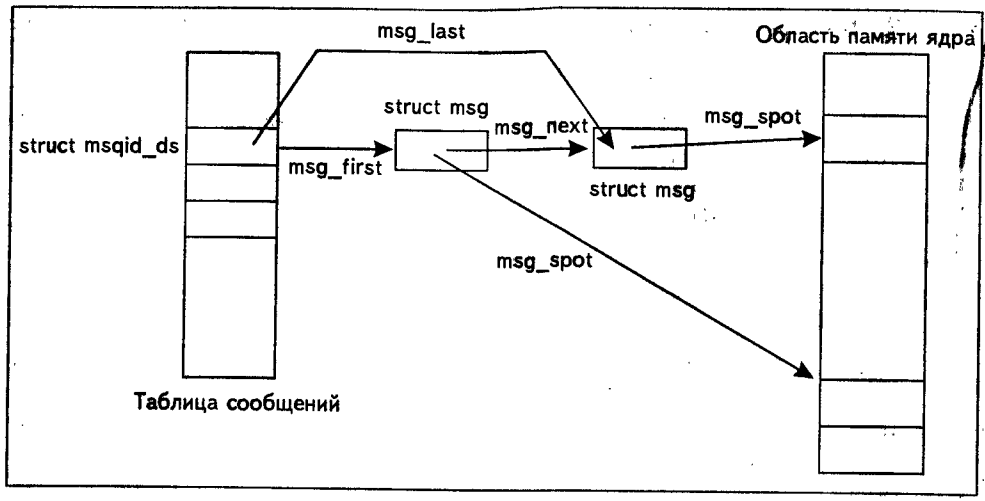


Рис. 10.2. Типы данных для очереди сообщений и записей сообщений

Для манипулирования сообщениями используются четыре API:

API сообщений	Назначение
msgget	Открытие для доступа и создание (при необходимости) очереди сообщений
msgsnd	Передача сообщения в очередь
msgrcv	Прием сообщения из очереди
msgctl	Манипулирование управляющими параметрами очереди сообщений

Как отмечалось выше, сообщения обрабатываются так же, как файлы UNIX, поэтому между API сообщений и API файлов можно провести следующую аналогию.

API сообщений	API файлов
msgget	open
msgsnd	write
msgrcv	read
msgctl	stat, unlink, chmod, chown

Для этих API сообщений необходимы следующие файлы заголовков:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```


10.3.3. Функция msgget

Прототип функции *msgget* выглядит следующим образом:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/message.h>

int msgget (key_t key, int flag );
```

Эта функция открывает очередь сообщений, идентификатор которой совпадает со значением аргумента *key*, и возвращает положительный целочисленный дескриптор. Его можно использовать в других API сообщений для передачи и приема сообщений, а также для запроса и/или установки управляющих параметров очереди.

Если значение аргумента *key* — положительное целое, этот API пробует открыть очередь сообщений, идентификатор которой совпадает с данным значением. Если же значением *key* является макрос IPC_PRIVATE, API создает новую очередь сообщений, которая будет использоваться исключительно вызывающим процессом.

Если аргумент *flag* имеет нулевое значение и нет очереди сообщений, идентификатор которой совпадал бы с заданным значением *key*, то API прерывается; в противном случае возвращается дескриптор этой очереди. Если процессу необходимо создать новую очередь (когда нет ни одной очереди), то значение аргумента *flag* должно содержать макрос IPC_CREAT, а также права на чтение сообщений из новой очереди и запись в нее (для владельца, группы и прочих).

Например, следующий системный вызов создает очередь сообщений с ключевым идентификатором 15 и правами доступа 0644 (чтение-запись для владельца, только чтение для членов группы и прочих пользователей), если такая очередь не существует. Этот вызов возвращает целочисленный дескриптор для последующих обращений к данной очереди:

```
int msgfdesc = msgget ( 15, IPC_CREAT|0644 );
```

Если желательно иметь гарантию создания новой очереди сообщений, можно указать одновременно с флагом IPC_CREAT флаг IPC_EXCL, и этот API будет успешно выполнен только в том случае, если он создаст новую очередь с заданным значением *key*.

В случае неудачи такой API возвращает -1. Вот некоторые возможные причины неудачи:

Значение *errno*	Причина ошибки
ENOSPC	Достигнут установленный системой лимит MSGMNI
ENOENT	Аргумент *flag* не содержит флаг IPC_CREAT, а очереди с указанным значением *key* нет

Значение *errno*	Причина ошибки
EEXIST	В аргументе *flag* установлены флаги IPC_EXCL и IPC_CREAT, а очередь с указанным значением *key* уже есть
EACCESS	Очередь с указанным значением *key* существует, но у вызывающего процесса нет прав доступа к ней

10.3.4. Функция msgsnd

Прототип функции *msgsnd* выглядит следующим образом:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (int msgfd, const void* msgPtr, int len, int flag );
```

Этот API передает сообщение (на которое указывает *msgPtr*) в очередь, обозначенную дескриптором *msgfd*.

Значение *msgfd* определяется возвращаемым значением функции *msgget*.

Фактическое значение аргумента *msgPtr* — указатель на объект, который содержит реальный текст и тип сообщения, подлежащего передаче. Для объявления объекта в этом случае можно использовать следующий тип данных:

```
struct msgbuf
{
   long mtype;              // тип сообщения
   char mtext[MSGMAX];      // буфер для текста сообщения
};
```

Значение аргумента *len* — это размер (в байтах) поля *mtext* объекта, на который указывает аргумент *msgPtr*.

Аргумент *flag* может иметь значение 0. Это означает, что при необходимости процесс можно блокировать до тех пор, пока данная функция не будет успешно выполнена. Если этот аргумент имеет значение IPC_NOWAIT, то при блокировании процесса выполнение функции прерывается.

В случае успешного выполнения этот API возвращает 0, а в случае неудачи — -1.

Ниже приведена программа *test_msgsnd.C*, которая создает новую очередь сообщений с ключевым идентификатором 100 и устанавливает следующие права доступа к очереди: чтение и запись для владельца, только чтение для членов группы, только запись для прочих пользователей. Если вызов *msgget* выполнен успешно, процесс передает в эту очередь сообщение *Hello*, тип которого — 15, и указывает, что данный вызов — неблокирующий. Если *msgget* или *msgsnd* дают сбой, процесс вызывает функцию *perror*, которая выводит на экран диагностическое сообщение.


```
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#ifndef MSGMAX
#define MSGMAX    1024
#endif

struct mbuf
{
   long mtype;
   char mtext[MSGMAX];
} mobj = { 15, "Hello" };


int main()
{
   int fd = msgget (100, IPC_CREAT|IPC_EXCL|0642);
   if (fd==-1 || msgsnd(fd,&mobj,strlen(mobj.mtext)+1, IPC_NOWAIT))
   perror("message");
   return 0;
}
```

10.3.5. Функция msgrcv

Прототип функции *msgrcv* выглядит так:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (int msgfd, const void* msgPtr, int len, int mtype, int flag );
```

Этот API принимает сообщение типа *mtype* из очереди сообщений, обозначенной дескриптором *msgfd*. Полученное сообщение хранится в объекте, на который указывает аргумент *msgPtr*. Значение аргумента *len* — это максимальный размер (в байтах) текста сообщения, которое может быть принято данным вызовом.

Значение *msgfd* берется из вызова функции *msgget*.

Фактическое значение аргумента *msgPtr* — указатель на объект, имеющий структуру данных, похожую на *struct msgbuf*.

Значение *mtype* — это тип сообщения, подлежащего приему. Ниже перечислены возможные значения данного аргумента и их смысл.

Значение *mtype*	Смысл
0	Принять из очереди самое старое сообщение любого типа
Положительное целое	Принять самое старое сообщение указанного типа
Отрицательное целое	Принять сообщение, тип которого меньше абсолютного значение *mtype* или равен ему. Если таких сообщений в очереди несколько, принять то, которое является самым старым и имеет наименьшее значение типа

Аргумент *flag* может иметь значение 0. Это означает, что процесс можно блокировать, если ни одно сообщение в очереди не удовлетворяет критериям выбора, заданным аргументом *mtype*. Если в очереди есть сообщение, которое удовлетворяет этим критериям, но превышает величину *len*, то функция возвращает код неудачного завершения.

Если процесс указал в аргументе *flag* значение IPC_NOWAIT, то вызов будет неблокирующим. Кроме того, если в названном аргументе установлен флаг MSG_NOERROR, то сообщение, находящееся в очереди, можно читать (даже если его размер превышает *len* байтов). Функция возвращает вызывающему процессу первые *len* байтов текста сообщения, а остальные данные отбрасывает.

Функция *msgrcv* возвращает количество байтов, записанных в буфер *mtext* объекта, на который указывает аргумент *msgPtr*, или -1 (в случае неудачи).

Приведенная ниже программа *test_msgrcv.C* подобна программе, рассмотренной в предыдущем разделе, но после передачи сообщения в очередь процесс вызывает API *msgrcv*, который ждет и выбирает из очереди сообщение типа 20. Если вызов выполняется успешно, процесс направляет выбранное сообщение на стандартное устройство вывода; в противном случае он вызывает функцию *perror*, которая выводит на экран диагностическое сообщение.

```
#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#ifndef MSGMAX
#define MSGMAX     1024
#endif

/* структура данных сообщения */
struct mbuf
{
   long mtype;
   char  mtext[MSGMAX];
} mobj = ( 15, "Hello" );
```



```
int main()
{
    int perm = S_IRUSR|S_IWUSR|S_IRGRP|S_IWOTH;
    int fd = msgget (100, IPC_CREAT|IPC_EXCL|perm);

    if (fd==-1 || msgsnd(fd,&mobj,strlen(mobj.mtext)+1,IPC_NOWAIT))
      perror("message");
    else if (msgrcv(fd,&mobj, MSGMAX,0,IPC_NOWAIT|MSG_NOERROR)0)
      cout << mobj.mtext << endl;
    else perror("msgrcv");
      return 0;
}
```

10.3.6. Функция msgctl

Прототип функции *msgctl* имеет следующий вид:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (int msgfd, int cmd struct msqid_ds* mbufPtr );
```

С помощью этого API можно запрашивать управляющие параметры очереди сообщений, обозначенной аргументом *msgfd*, изменять информацию в управляющих параметрах очереди, удалять очередь из системы.

Значение аргумента *msgfd* берется из вызова функции *msgget*.

Ниже перечислены возможные значения аргумента *cmd* и их смысл.

Значение *cmd*	Смысл
IPC_STAT	Копировать управляющие параметры очереди в объект, указанный аргументом *mbufPtr*
IPC_SET	Заменить управляющие параметры очереди параметрами, содержащимися в объекте, на который указывает аргумент *mbufPtr*. Чтобы выполнить эту операцию, вызывающий процесс должен иметь права либо привилегированного пользователя, либо создателя или назначенного владельца очереди. С помощью этого API можно устанавливать UID владельца очереди и идентификатор его группы, права доступа и (или) уменьшать лимит *msg_qbyte* очереди
IPC_RMID	Удалить очередь из системы. Чтобы выполнить эту операцию, вызывающий процесс должен иметь права либо привилегированного пользователя, либо создателя или назначенного владельца очереди

В случае успешного выполнения этот API возвращает 0, в случае неудачи — -1.

Приведенная ниже программа *test_msgctl.C* "открывает" очередь сообщений с ключевым идентификатором 100 и вызывает функцию *msgctl* для

считывания управляющих параметров очереди. Если вызовы *msgget* и *msgctl* выполняются успешно, процесс выводит на экран количество сообщений, находящихся в очереди на данный момент времени. Еще одним вызовом *msgctl* процесс устанавливает идентификатор владельца очереди равным своему идентификатору. Наконец, вызвав *msgctl* в третий раз, процесс удаляет очередь.

```
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int main()
{
    struct msqid_ds mbuf;
    int fd = msgget (100, 0);
    if (fd0 && msgctl(fd,IPC_STAT,&mbuf))  {
      cout << "#msg in queue: " << mbuf.msg_qnum << endl;
      mbuf.msg_perm.uid = getuid();  // изменить UID владельца
      if (msgctl(fd,IPC_SET,&mbuf)==-1)
          perror("msgctl");
   } else
      perror("msgctl");
   if (msgctl(fd,IPC_RMID,0)) perror("msgctl - IPC_RMID");
   return 0;
}
```

10.3.7. Пример приложения клиент/сервер

В этом разделе описывается приложение клиент/сервер, в котором используются сообщения. Первая программа, *server.C*, создает серверный процесс-демон, который непрерывно работает в фоновом режиме, предоставляя услуги клиентским процессам. Клиентские процессы создаются программой *client.C*. Они отправляют процессу-демону запросы на обслуживание, передавая сообщения в очередь, которой владеет и управляет этот демон. Демон отвечает клиентам, передавая сообщения в эту же очередь.

В частности, каждое сообщение-запрос на обслуживание, передаваемое клиентским процессом серверу-демону, должно иметь следующие поля:

Поле данных сообщения	Содержание
message type (тип сообщения)	Целое число, определяющее, какой командой осуществляется запрос на обслуживание
message text (текст сообщения)	Идентификатор клиентского процесса, представленный в виде строки символов (данные, хранящиеся в этом поле, могут быть любым произвольным потоком байтов, содержащим в том числе и непечатаемые символы)

Сервером поддерживаются следующие команды запроса на обслуживание:

Команда запроса на обслуживание	Предоставляемая услуга
1	Посылает клиенту значения местного времени и даты
2	Сообщает клиенту текущее время в универсальном формате (UCT)
3	Удаляет очередь сообщений и завершает процесс-демон. Ответ клиенту не посылается
4-99	Посылает клиенту сообщение об ошибке

Каждое ответное сообщение, посылаемое сервером клиенту, содержит следующие поля:

Поле данных сообщения	Содержание
message type (тип сообщения)	Идентификатор клиентского процесса
message text (текст сообщения)	Данные ответа сервера в виде строки символов

Поскольку сервер и клиент взаимодействуют между собой через общую очередь сообщений, то можно определить класс *message*, инкапсулирующий все API, предназначенные для обмена сообщениями. Этот класс определяется в следующем заголовке *message.h*, который используется обеими программами, *server.C* и *client.C*:

```
#ifndef MESSAGE_H
#define MESSAGE_H
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/wait.h>

/* общие объявления для серверного процесса */
enum { MSGKEY= 176, MAX_LEN =256, ILLEGAL_CMD = 4 );
enum { LOCAL_TIME = 1, UTC_TIME = 2, QUIT_CMD = 3};

typedef struct mgbuf
{
      long   mtype;
      char   mtext[MAX_LEN];
} MSGBUF;

class message
{
   private:
```

```
  int    msgId;                  // дескриптор очереди сообщений
  struct mgbuf mObj;
public:
  /* конструктор. Получаем указатель на очередь сообщений.*/
  message ( int key )
  {
     if (msgId=msgget(key,IPC_CREAT|0666)==-1)
         perror("msgget");
  };

  /* функция-деструктор. В данном случае ничего не делает */
   ~message () ();

  /* проверить, открыта или нет очередь сообщений */
  int good () ( return (msgId >= 0) ? 1 : 0; };

  /* удалить очередь сообщений */
  int rmQ ()
  {
     int rc=msgctl(msgId,IPC_RMID,0);
     if (rc==-1) perror("msgctl");
     return rc;
  };

  /* передать сообщение */
  int send ( const void* buf, int size, int type)
  {
     mObj.mtype = type;
     memcpy(mObj.mtext,buf,size);
     if (msgsnd(msgId,&mObj,size,0)) {
         perror("msgsnd"); return -1;
     );
     return 0;
  );

  /* принять сообщение */
  int rcv ( void* buf, int size, int type, int* rtype)
  {
     int len = msgrcv(msgId,&mObj,MAX_LEN,type,MSG_NOERROR);
      if (len==-1) (
        perror("msgrcv"); return -1;
     )
      memcpy(buf,mObj.mtext,len);
     if (rtype) *rtype = mObj.mtype;
     return len;
  );
);
#endif
```

Преимущество применения класса *message* состоит в том, что прикладным программам не нужно использовать базовые API сообщений. Манипулируя типом сообщения и буфером сообщений, они взаимодействуют с

объектом этого класса, применяя практически тот же самый интерфейс, который API *read* и *write* используют для файлов. Таким образом можно сократить затраты времени на программирование. Более того, в следующем разделе будет показано, как можно изменить класс *message* для реализации других методов IPC, не корректируя при этом код приложения.

На примере приведенной ниже программы *server.C* показано использование заголовка *message.h*:

```
#include <strstream.h>
#include "message.h"
#include <string.h>
#include <signal.h>

int main()
{
  int      len, pid, cmdId;
  char     buf[256];
  time_t   tim;

  /* настроить этот процесс как демон */
  for (int i=0; i < 20; i++) signal(i,SIG_IGN);
  setsid();
  cout <<"server: start executing...\n"<<flush;
  message mqueue(MSGKEY);        // открыть очередь сообщений
  if (!mqueue.good()) exit(1);   // выйти из программы, если очередь
                                 // не открывается

  /* ожидать запрос от клиента */
  while (len=mqueue.rcv(buf,sizeof(buf),-99,&cmdId) > 0)
  {
     /* получить идентификатор процесса клиента и проверить его
        достоверность */
     istrstream(buf,sizeof(buf))>>pid;
     if (pid < 100)
     {
       cerr << "Illegal PID:"<<buf<<"/"<<pid<< endl;
        continue;
     }
     /* подготовить ответ клиенту */
      cerr<<"server: receive cmd #"<<cmdId
          <<",from client:"<<pid<<endl;
     switch (cmdId)
     {
        case LOCAL_TIME:
          tim = time(0);
           strcpy(buf,ctime(&tim));
          break;
        case UTC_TIME:
          tim = time(0);
           strcpy(buf,asctime(gmtime(&tim)));
          break;
```

```
        case QUIT_CMD:
          cerr <<"server: deleting msg queue...\n";
          return mqueue.rmQ();
       default:          /* послать клиенту сообщение об ошибке */
          ostrstream(buf,sizeof(buf))<<"Illegal cmd:"<< cmdId <<'\0';
     }
     /* послать клиенту ответ */
     if (mqueue.send(buf,strlen(buf)+1,pid)==-1)
     cerr<<"Server: "<<getpid()<<"send response fails\n";
   }; /* бесконечный цикл */
   return 0;
}
```

Начинается этот серверный процесс с того, что настраивает себя как демон: игнорирует все основные сигналы и делает себя лидером сеанса и группы процессов. Теперь он не зависит от своего родительского процесса.

Затем серверный процесс создает объект-сообщение с ключевым идентификатором 176 (он выбран произвольно). Функция-конструктор класса создает очередь сообщений, если она еще не существует, присваивает ей ключевой идентификатор и предоставляет права на чтение и запись всем пользователям.

После создания объекта-сообщения сервер входит в цикл опроса и ожидает передачи клиентскими процессами в этот объект сообщений-запросов на обслуживание. В частности, поскольку типы клиентских команд запроса на обслуживание ограничены диапазоном 1—99, сервер опрашивает сообщения, тип которых не превышает 100. Прочитав запрос на обслуживание, сервер проверяет, превышает ли идентификатор процесса клиентского запроса значение 99, а затем передает клиенту ответное сообщение на основании полученной команды запроса на обслуживание. Если команда обслуживания — QUIT_CMD, сервер удаляет объект-сообщение и завершается.

При разработке программы, в которой используются сообщения, следует помнить о том, что у процесса, как правило, редко возникает необходимость чтения посланных им же сообщений. Важно поэтому, чтобы процесс использовал для отправляемых и получаемых сообщений разные наборы типов сообщений. В рассматриваемом примере типами ответных сообщений сервера являются идентификаторы процессов клиентов. Считается, что эти идентификаторы всегда больше или равны 100 (данное ограничение вводится в программе *client.C*). Кроме того, типами клиентских сообщений-запросов на обслуживание являются типы команд запроса на обслуживание (значения этих типов находятся в диапазоне от 1 до 99). Таким образом, сервер читает только клиентские сообщения-запросы на обслуживание, но не ответные сообщения на них, а клиенты читают только ответные сообщения сервера, но не свои собственные сообщения-запросы на обслуживание.

Ниже приведен текст программы *client.C*.

```
#include <strstream.h>
#include <string.h>
#include "message.h"

int main()
{
   int   cmdId, pid;
   while (getpid() < 100)
       switch (pid=fork())  {    // PID клиентского процесса > 99
         case -1:   perror("fork"), exit(1);
         case 0:    break;
         default:   waitpid(pid,0,0); exit(0);
       }
     cout << "client: start executing...\n";
     message mqueue(MSGKEY);          // создать объект-сообщение
     if (!mqueue.good()) exit(1);     // выйти из программы, если
                                      // очередь не открывается
     char procId[256], buf2[256];
     ostrstream(procId,sizeof(procId)) << getpid() << '\0';
     do  {
       /* прочитать команду со стандартного ввода */
       cout << "cmd> " << flush; // вывести на экран приглашение на ввод
       cin >> cmdId; // прочитать команду, поступившую от пользователя
       cout << endl;                  // ввести с консоли <CR>
       if (cin.eof()) break;          // выйти из программы, если
                                      // встретился символ EOF

       /* проверить допустимость команды */
       if (!cin.good() || cmdId < 0 || cmdId > 99) {
          cerr << "Invalid input: " << cmdId << endl; continue;
       }
       /* передать запрос в демон */
       if (mqueue.send(procId,strlen(procId),cmdId))
           cout << "client: " << getpid() << " msgsnd error\n";
       else if (cmdId==QUIT_CMD) break;      /* при QUIT_CMD выйти */
       /* принять данные от демона */
       else if (mqueue.rcv(buf2,sizeof(buf2),getpid(),0)==-1)
           cout << "client: " << getpid() << " msgrcv error\n";

       /* вывести на экран ответ сервера */
       else cout << "client: " << getpid() << "  " << buf2 << endl;

     } while (1);  /* цикл до EOF */

     cout << "client: " << getpid() << " exiting...\n" << flush;
     return 0;
}
```

Эта программа-клиент прежде всего проверяет идентификатор процесса, который всегда больше или равен 100. Если это не так, она рекурсивно

вызывает функцию *fork* до тех пор, пока один из идентификаторов порожденных процессов не будет больше или равен 100. При этом все не удовлетворяющие этому условию родительские процессы просто ждут, пока их порожденный процесс завершится, а затем завершаются сами. Этот простой способ гарантирует соблюдение условия взаимодействия по схеме клиент/сервер (идентификатор клиентского процесса должен быть больше 99).

После создания клиентского процесса он открывает объект-сообщение, имеющий тот же ключевой идентификатор, что и сервер. Затем программа начинает цикл, в котором приглашает пользователей пошагово ввести со стандартного устройства ввода команду запроса на обслуживание. Для каждой принимаемой команды программа проверяет принадлежность значения ее типа диапазону 1—99, а затем посылает в серверный процесс сообщение-запрос на обслуживание. После этого программа читает ответ сервера и направляет поступившие данные на стандартное устройство вывода.

Клиентский процесс завершается, когда в данных, поступивших на стандартное устройство ввода, встречается признак конца файла (например, пользователь нажимает клавиши [Ctrl+D]) или когда при чтении данных со стандартного устройства ввода возникает ошибка.

Вот пример взаимодействия этих программ:

```
chp13 % server &
server: start execution...
[1] 356
chp13 % client
client: start execution...
cmd> 1
server: receive cmd #1, from client: 357
client: 357 Tue Jan 24 22:23:17 1995
cmd> 2
server: receive cmd #2, from client: 357
client: 357 Wed Jan 25 06:23:19 1995
cmd> 4
server: receive cmd #4, from client: 357
client: 357 Illegal cmd: 4
cmd> 3
client: 357 exiting...
server: receive cmd #3, from client: 357
server: deleting msg queue...
[1] Done    server
chp13%
```

В приведенных выше примерах с сервером взаимодействует только один клиент, но могут взаимодействовать и несколько клиентских процессов одновременно.

10.4. Сообщения в стандарте POSIX.1b

Сообщения POSIX.1b создаются и обрабатываются приблизительно так же, как сообщения UNIX System V. В частности, в стандарте POSIX.1b определяются заголовок <mqueue.h> и набор API сообщений:

```
#include <mqueue.h>

mqd_t mq_open ( char* name, int flags, mode_t mode,
                                        struct mq_attr* attrp );

int mq_send ( mqd_t mqid, const char* msg, size_t len, unsigned priority );
int mq_receive ( mqd_t mqid, char* buf, size_t len, unsigned* prio );
int mq_close ( mqd_t mqid );
int mq_notify ( mqd_t mqid, const struct mq_sigevent* sigvp);
int mq_getattr ( mqd_t mqid, struct mq_attr* atttrp );
int mq_setattr ( mqd_t mqid, struct mq_attr* atttrp, struct mq_attr* oattrp );
```

API *mq_open* похож на функцию *msgget*: он возвращает идентификатор очереди сообщений типа *mqd_t*. Отметим, что идентификатор *mqd_t* не является целочисленным дескриптором.

Очереди сообщений, которую открывает функция *mq_open*, присваивается имя, указанное в аргументе *name*. Значение *name* должно быть строкой символов, похожей на путевое UNIX-имя, и всегда начинаться с символа "/". Допустимость использования дополнительных символов "/" зависит от варианта системы. Кроме того, пользователям не следует ожидать, что в результате этого вызова будет создан файл с таким же именем.

Аргумент *flags* задает способ доступа к очереди со стороны вызывающего процесса. Он может иметь следующие значения: O_RDONLY (вызывающий процесс может только принимать сообщения); O_WRONLY (вызывающий процесс имеет право лишь передавать сообщения); O_RDWR (вызывающий процесс может передавать и принимать сообщения). Если указан флаг O_CREAT, то в случае отсутствия заданной очереди она будет создана. Выполнение функции будет прервано, если за флагом O_CREAT следует флаг O_EXCL, но указанная очередь уже существует.

Наконец, можно присвоить переменной *oflag* значение O_NONBLOCK, которое указывает, что доступ к очереди сообщений (осуществляемый посредством API *mq_send* и *mq_receive*) будет неблокирующим.

Аргументы *mode* и *attrp* нужны только в том случае, если в аргументе *oflag* указан флаг O_CREAT. Они задают для очереди сообщений атрибуты и разрешения на чтение и запись. Тип данных *struct mq_attr* определяется в заголовке <mqueue.h>.

В случае успешного выполнения эта функция возвращает идентификатор очереди сообщений типа *mqd_t*, а в случае неудачи — значение (*mqd_t*)-1.

В следующем примере показано, как открывается и создается (при необходимости) очередь сообщений */foo* с доступом для чтения и записи. Разрешение на чтение-запись для вновь создаваемой очереди устанавливается только по категории "владелец". В новой очереди может находиться до 200 сообщений, длина каждого из которых не должна превышать 1024 байтов...

```
struct mq_attr attrv;        // содержит атрибуты для новой очереди
attrv.mq_maxmsg = 200;       // в очереди может находиться максимум
                             // 200 сообщений
attrv.mq_msgsize = 1024;     // максимум 1024 байта в одном сообщении
attrv.mq_flags = 0;

mqd_t mqid = mq_open("/foo", O_RDWR | O_CREAT, S_IRWXU, &attrv);
if (mqid == (mq_t)-1) perror("mq_open");
```

API *mq_send* передает сообщение в очередь, указанную аргументом *mqid*. Значение аргумента *msg* — это адрес буфера, который содержит текст сообщения, аргумент *len* задает размер текста сообщения (количество символов). Значение *len* должно быть меньше лимита, установленного для данной очереди сообщений (максимального размера сообщения). В противном случае вызов не будет выполнен.

Значение аргумента *priority* — это целое число из диапазона от 0 до MQ_PRIO_MAX. Оно используется для сортировки сообщений, имеющихся в очередях, при этом сообщения с более высоким значением *priority* обрабатываются в первую очередь. Если у двух и более сообщений одинаковое значение *priority*, они сортируются в порядке убывания их времени нахождения в очереди — старые сообщения извлекаются раньше, чем новые.

В случае успешного выполнения эта функция возвращает 0, а в случае неудачи — -1. Отметим, что данная функция может блокировать вызывающий процесс, если очередь сообщений уже полна. Если же очередь была открыта с флагом O_NONBLOCK, функция прерывается и немедленно возвращает код неудачного завершения, -1.

В следующем примере сообщение *Hello POSIX.1b* передается в очередь сообщений, указанную аргументом *mqid*, с приоритетом 5:

```
char* msg = "Hello POSIX.1b";
if (mq_send(mq_id, msg, strlen(msg)+1, 5)) perror("mq_send");
```

API *mq_receive* принимает самое старое и имеющее наивысший приоритет сообщение из очереди сообщений, указанной аргументом *mqid*. Значение аргумента *buf* — адрес буфера, который содержит текст сообщения, аргумент *len* задает максимальный размер буфера *buf*. Если размер принимаемого сообщения больше *len* байтов, эта функция возвращает код неудачного завершения.

Аргумент *prio* — это адрес целочисленной беззнаковой переменной, содержащей значение приоритета принимаемого сообщения. Если значение этого аргумента задано как NULL, то приоритет принимаемого сообщения роли не играет.

В случае успешного выполнения данная функция возвращает количество байтов текста сообщения, помещенных в буфер *buf*, а в случае неудачи — -1. Если очередь сообщений пуста, функция может заблокировать вызывающий процесс. В ситуации же, когда очередь была открыта с флагом O_NONBLOCK, функция прерывается и немедленно возвращает код -1.

Если вызов *mq_receive* блокирует несколько процессов, то прибывшее в очередь сообщение будет получено процессом, имеющим наивысший приоритет и дольше других ожидающим поступления сообщения.

В следующем примере принимается сообщение из очереди, указанной аргументом *mqid*:

```
char buf[256];
unsigned prio;
if (mq_receive( mqid, buf, sizeof buf, &prio )== -1)
perror("mq_receive");
cerr << "receive msg:" << buf << ", priority=" << prio <<endl;
```

API *mq_close* освобождает ресурсы, использованные при открытии очереди сообщений с заданным идентификатором сообщения, *mqid*. В случае успешного выполнения эта функция возвращает 0, а в случае неудачи — -1.

API *mq_notify* используется в том случае, если процесс должен получить асинхронное уведомление о прибытии сообщения в пустую очередь, а не блокироваться вызовом *mq_receive* в ожидании такого события. Аргумент *mqid* указывает очередь, в которой нужно контролировать прибытие сообщений. Значение аргумента *sigvp* — это адрес переменной типа *struct sigevent*. Тип данных *struct sigevent* определяется в заголовке <signal.h> и содержит номер сигнала, который должен генерироваться вызывающим процессом при прибытии сообщения в указанную очередь.

Если существует процесс, который уже получил уведомление или заблокирован вызовом *mq_receive*, данная функция не выполняется. Более того, даже если процесс успешно выполнил вызов *mq_notify* и сигнал уведомления доставлен (поскольку сообщение поступило в очередь), он все равно может не принять сообщение, если какой-то другой процесс вызовет *mq_receive* раньше.

Наконец, если значение аргумента *sigvp* задано как NULL, вызов *mq_notify* отменяет отправку уведомления для указанной очереди сообщений. Такое уведомление не отправляется также в том случае, если процесс уже существует или вызывает для данной очереди сообщений API *mq_close*.

В случае успешного выполнения эта функция возвращает 0, а в случае неудачи — -1.

В следующем примере регистрируется сигнал SIGUSR1, который должен быть доставлен в вызывающий процесс в случае прибытия сообщения в очередь, указанную переменной *mqid*:

```
struct sigevent sigv;
sigv.sigev_notify = SIGEV_SIGNAL; // запрос уведомления
sigv.sigev_signo = SIGUSR1;       // послать SIGUSR1 для уведомления
if (mq_notify(mqid, &sigv)== -1) perror("mq_notify");
```

API *mq_getattr* запрашивает атрибуты очереди сообщений, указанной аргументом *mqid*. Аргумент *attrp* — это адрес переменной типа *struct mq_attr*. Тип данных *struct mq_attr* определяется в заголовке <mqueue.h>. Вот некоторые полезные поля этой структуры.

Поле	Содержание
mq_flags	Показывает, является ли выполняемая над очередью операция блокирующей. Возможное значение — 0 или O_NONBLOCK
mq_maxmsg	Максимальное число сообщений, которое может находиться в очереди в любой данный момент времени
mq_msgsize	Максимальный размер сообщения в байтах
mq_curmsgs	Число сообщений, находящихся в очереди в данный момент времени

В случае успешного выполнения эта функция возвращает 0, а в случае неудачи — -1.

В следующем примере программа получает информацию об атрибутах очереди сообщений, указанной переменной *mqid*:

```
struct mq_attr_attrv;
if (mq_getattr(mqid, &attrv)==-1)
   perror("mq_getattr");
else cout << "flags =" << attrv.mq_flags
          << ", cur. no. msg"
          << attrv.mq_curmsgs << endl;
```

API *mq_setattr* устанавливает атрибут *mq_attr::mq_flags* для очереди сообщений, указанной аргументом *mqid*. Аргумент *attrp* — это адрес переменной типа *struct mq_attr*. API *mq_setattr* использует только значение *attrp->mq_flags*. Допустимое значение этого поля — 0 (блокирующая операция над очередью) или O_NONBLOCK (неблокирующая операция над очередью). С помощью аргумента *oattrp*, задающего адрес переменной типа *struct mq_attr*, возвращается та же информация, что и при вызове *mq_getattr*.

В случае успешного выполнения эта функция возвращает 0, а в случае неудачи — -1.

В следующем примере мы задаем выполнение неблокирующих операций над очередью сообщений, обозначенной переменной *mqid*. Старые атрибуты очереди игнорируются.

```
struct mq_attr atrrv;
attrv.mq+flags = O_NONBLOCK;
if (mq_setattr(mqid, &attrv, 0}==-1} perror("mq_setattr"};
```

10.4.1. Класс message стандарта POSIX.1b

Класс сообщений, определенный в разделе 10.3.7, предназначен для обработки сообщений только ОС UNIX System V. В следующем заголовке *message2.h* определяется новый класс сообщений, в котором используются

API сообщений стандарта POSIX.1b. Следует отметить, что интерфейс этого нового класса идентичен интерфейсу, приведенному в разделе 10.3.7:

```
#ifndef MESSAGE2_H
#define MESSAGE2_H
#include <stdio.h.>
#include <memory.h>
#include <sys/ipc.h>
#include <mqueue.h>          // использовать API сообщений POSIX.1b

/* общие объявления для серверного процесса */
enum { MSGKEY=186, MAX_LEN=256, ILLEGAL_CMD = 4};
enum { LOCAL_TIME = 1, UTC_TIME = 2, QUIT_CMD = 3 };
struct mgbuf
{
      long        mtype;
      char        mtext[MAX_LEN];
};

/* класс сообщений POSIX.1b */
class message
{
   private:
    mqd_t    msgId;        // идентификатор очереди сообщений
    struct mgbuf mObj;
   public:
     /* функция-конструктор, совместимая с System V */
     message( int key }
     {
       char name[80};
       sprintf(name,"/MQUEUE%d",key);
       if ((msgId=mq_open(name,O_RDWR|IPC_CREAT,0666,0)}==(mqd_t}-1}
          perror("mq_open");
     };

     /* функция-конструктор, соответствующая POSIX.1b */
     message( const char* name }
     {
       if ((msgId=mq_open(name,O_RDWR|IPC_CREAT,0666,0)}==(mqd_t)-1)
          perror("mq_open");
     };

     /* функция-деструктор */
     ~message() {  (void}mq_close( msgId}; };

     /* проверить статус открытия очереди */
     int good() { return (msgId >= 0} ? 1 : 0; };
     /* удалить очередь сообщений */
     int rmQ()
     {
       return mq_close( msgId );
     };
```

```
    /* передать сообщение */
    int send( const void* buf, int size, int type)
    {
       mObj.mtype = type;
       memcpy(mObj.mtext,buf,size);
       if (mq_send(msgId,(char*)&mObj,size,type)) {
          perror("mq_send");
          return -1;
       };
       return 0;
    );

    /* принять сообщение */
    int rcv( void* buf, int size, int type, unsigned* rtype)
    (
       struct mq_attr attrv;
       if (mq_getattr(msgId,&attrv)==-1) {
          perror("mq_getattr");
          return -1;
        )
        if (!attrv.mq_curmsgs) return -1;     // сообщений нет

        int len = mq_receive(msgId,(char*)&mObj,MAX_LEN,rtype);
        if (len < 0) {
          perror("mq_receieve");
          return -2;
       )

       if (type && ((type > 0 && type!=*rtype) ||
                    (type < 0 && -type < *rtype))) {
         mq_send(msgId, (char*)&mObj, len, *rtype );
         return -3;                          // не тот тип
       }
        memcpy(buf,mObj.mtext,len);
       return len;
     };
);
#endif          /* MESSAGE2_H */
```

Версия класса *message*, определенная в стандарте POSIX.1b, может использоваться так же, как версия, принятая в System V. В частности, есть две функции-конструктора *message::message*, одна из которых вызывается с целочисленным ключом, а другая с именем сообщения. В любом случае для открытия и создания (в случае необходимости) очереди сообщений вызывается функция *mq_open*.

Функции *message::send*, *message::rcv* и *message::~message* имеют такой же интерфейс, что и соответствующие функции в классе *message* в System V. Единственное различие заключается в том, что функция *message::rcv* стандарта POSIX.1b не может выбирать сообщения на основании заданного пользователем типа. Эта функция получает из очереди сообщение с наивысшим

приоритетом. Если тип сообщения не соответствует типу, указанному пользователем, эта функция помещает сообщение обратно в очередь и возвращает -1. Тип сообщения соответствует типу, определенному пользователем, если:

1) заданный пользователем тип — 0 (т.е. тип сообщения значения не имеет);
2) определенный пользователем тип полностью совпадает с типом принятого сообщения;
3) указанный пользователем тип отрицателен, а абсолютная величина его целочисленного значения больше значения типа принятого сообщения или равна ему.

Заголовок *message2.h* можно использовать в тех же клиентской и серверной программах, которые описаны в разделе 13.3.7. Результаты выполнения скомпилированных клиентской и серверной программ должны совпадать с аналогичными результатами для версии System V.

10.5. Семафоры в UNIX System V

Семафоры — это средства синхронизации выполнения множества процессов. Семафоры группируются в наборы, каждый из которых содержит один и более семафоров. Семафоры часто используются совместно с разделяемой памятью, реализуя таким образом мощный метод межпроцессного взаимодействия.

API семафоров ОС UNIX System V выполняют следующие функции:

- создают набор семафоров;
- "открывают" набор семафоров и получают дескриптор, обозначающий данный набор;
- увеличивают и уменьшают целочисленные значения одного и более семафоров в наборе;
- запрашивают значения одного и более семафоров в наборе;
- запрашивают и устанавливают управляющие параметры набора семафоров.

Значением семафора является переменная типа unsigned short. Процесс, имеющий право на чтение семафоров, может запрашивать их значения. Процесс, имеющий право на изменение семафоров, может увеличивать и уменьшать их значения. Если процесс попытается уменьшить значение семафора так, чтобы оно стало отрицательным, эта операция и сам процесс будут заблокированы до тех пор, пока другой процесс не увеличит значение данного семафора до величины, достаточной для успешного выполнения операции заблокированного процесса (т.е. чтобы значение семафора после операции уменьшения было положительным или равным нулю). Этот принцип лежит в основе синхронизации процессов с помощью семафоров:

- процесс X, который должен дождаться выполнения процессом Y определенных действий, уменьшает значение одного или нескольких семафоров на некоторую величину, в результате чего блокируется ядром;

- когда процесс Y готов позволить процессу X возобновить выполнение, он увеличивает значения семафоров до величин, достаточных для успешного выполнения операции над семафором, осуществляемой процессом X. Ядро разблокирует процесс X.

Если значение семафора — положительное целое, то процесс, явно запрашивающий нулевое значение, блокируется до тех пор, пока другой процесс не уменьшит значение этого семафора до нуля.

Процесс, имеющий доступ к набору семафоров, может выполнять операции над отдельными семафорами или одновременно оперировать двумя и более семафорами. В последнем случае действует следующее правило: если какая-либо операция не может быть выполнена над любым из выбранных семафоров, то неудачной считается операция по отношению к обоим семафорам и их значения остаются без изменений. Это сделано для того, чтобы в случае, если два или более процессов попытаются прочитать и записать значения одного и того же набора семафоров, операции мог выполнить только один процесс.

10.5.1. Поддержка семафоров ядром UNIX

В ОС UNIX System V.3 и V.4 в адресном пространстве ядра имеется таблица семафоров, в которой отслеживаются все создаваемые в системе наборы семафоров. В каждом элементе таблицы семафоров находятся следующие данные об одном наборе семафоров:

- Имя, представляющее собой целочисленный идентификационный ключ, присвоенный набору семафоров процессом, который его создал. Другие процессы могут, указывая этот ключ, "открывать" набор и получать дескриптор для доступа к набору.
- UID создателя набора семафоров и идентификатор его группы. Процесс, эффективный UID которого совпадает с UID создателя, может удалять набор и изменять параметры управления им.
- UID назначенного владельца и идентификатор его группы. Эти идентификаторы обычно совпадают с идентификаторами создателя, но процесс создателя может устанавливать эти идентификаторы для переназначения владельца набора и принадлежности к группе.
- Права доступа к набору для чтения-записи по категориям "владелец", "группа" и "прочие". Процесс, имеющий право на чтение набора, может запрашивать значения семафоров и UID назначенного владельца и группы. Процесс, имеющий право на запись в набор, может изменять значения семафоров.
- Количество семафоров в наборе.
- Время изменения одного или нескольких значений семафоров последним процессом.
- Время последнего изменения управляющих параметров набора каким-либо процессом.

- Указатель на массив семафоров.

Семафоры в наборе обозначаются индексами массива: например, первый семафор в наборе имеет индекс нуль, второй — единицу и т.д. В каждом семафоре содержатся следующие данные:

- значение семафора;
- идентификатор процесса, который оперировал семафором в последний раз;
- число процессов, заблокированных в текущий момент и ожидающих увеличения значения семафора;
- число процессов, заблокированных в текущий момент и ожидающих обращения значения семафора в нуль.

На рис. 10.3 изображена структура данных ядра, используемая для манипулирования семафорами.

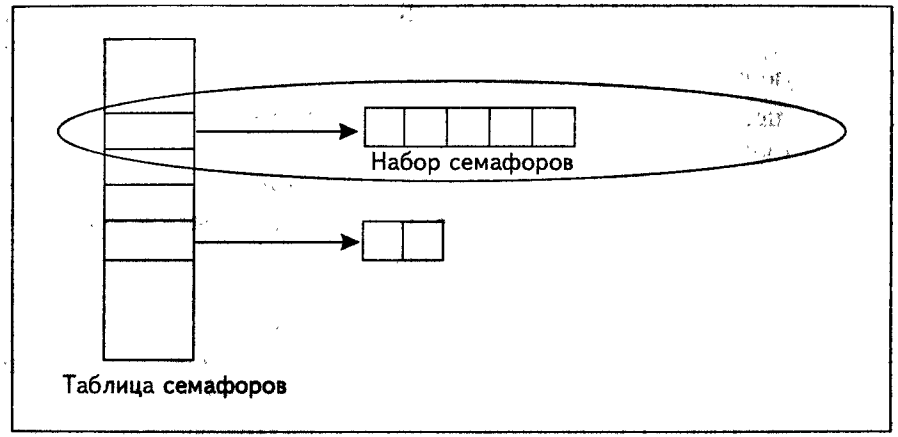


Рис. 10.3. Структура данных ядра, используемая для манипулирования семафорами

Как и сообщения, семафоры хранятся в адресном пространстве ядра и являются устойчивыми, т.е. сохраняются независимо от завершения создавшего их процесса. У каждого набора семафоров есть назначенный владелец, и удалить набор или изменить его управляющие параметры могут только процессы, имеющие права привилегированного пользователя, создателя набора или назначенного владельца. Если набор семафоров удаляется, то ядро активизирует все процессы, которые в данный момент заблокированы семафорами этого набора; все произведенные данными процессами системные вызовы прерываются и возвращают код неудачного завершения, -1.

Система устанавливает ряд ограничений на манипулирование семафорами. Они определяются в заголовке <sys/sem.h>:

Системное ограничение	Значение
SEMMNI	Максимальное число наборов семафоров, которые могут существовать в системе в любой данный момент времени
SEMMNS	Максимальное число семафоров во всех наборах, которые могут существовать в системе в любой данный момент времени
SEMMSL	Максимально допустимое число семафоров в одном наборе
SEMOPM	Максимальное число семафоров в наборе, над которыми могут одновременно выполняться операции в любой данный момент времени

Влияние этих системных ограничений на процессы таково:

- если попытка процесса создать новый набор семафоров вызовет превышение лимита SEMMNI или SEMMNS, этот процесс будет заблокирован до тех пор, пока какой-либо процесс не удалит один или несколько существующих наборов;
- если процесс попытается создать набор более чем из SEMMSL семафоров, этот системный вызов выполнен не будет;
- если процесс попытается одновременно выполнить операцию более чем над SEMOPM семафорами в наборе, этот системный вызов выполнен не будет.

10.5.2. API ОС UNIX для семафоров

В заголовке <sys/ipc.h> объявляются тип данных *struct ipc_perm*, который используется в данном наборе семафоров для хранения UID, GID, ID создателя и его группы, идентификатор и права доступа на чтение и запись.

Элементы таблицы семафоров имеют тип данных *struct semid_ds*, который определяется в заголовке <sys/sem.h>. Ниже перечислены названия полей этой структуры и данные, которые в них хранятся.

Поле	Данные
sem_perm	Данные, хранящиеся в записи *struct ipc_perm*
sem_nsems	Число семафоров в наборе
sem_base	Указатель на массив семафоров
sem_otime	Время, когда какой-либо процесс в последний раз выполнял операции над семафорами
sem_ctime	Время, когда тот или иной процесс в последний раз изменял управляющие параметры набора

Помимо этого, тип данных *struct sem*, определенный в заголовке <sys/sem.h>, используется для представления данных, хранящихся в семафоре.

Поле	Данные
semval	Целочисленное значение текущего семафора
sempid	Идентификатор процесса, который выполнял операции над данным семафором в последний раз
semncnt	Число процессов, которые заблокированы и ожидают увеличения значения семафора
semzcnt	Число процессов, которые заблокированы и ожидают обращения значения семафора в нуль

На рис. 10.4 показано, как вышеупомянутые структуры используются в таблице семафоров и в записях значений семафоров.

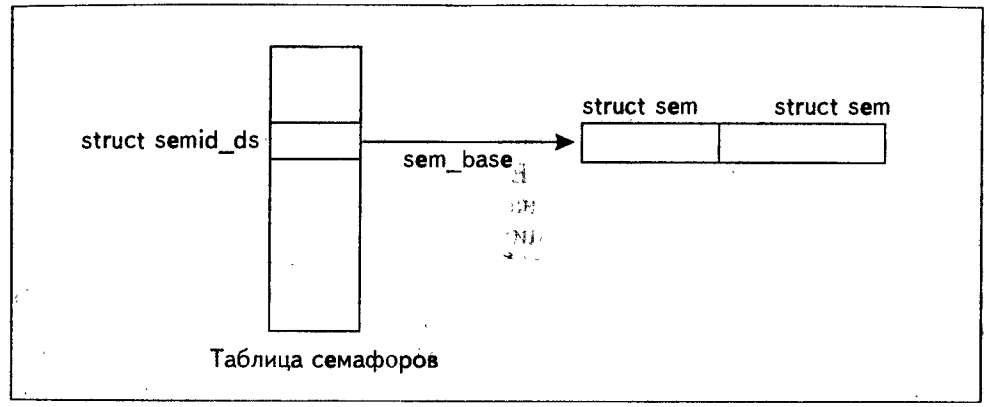


Рис. 10.4. Типы данных таблицы семафоров и записи значений семафора

В ОС UNIX System V имеются три API, предназначенные для манипулирования семафорами.

API семафоров	Назначение
semget	Открытие и создание (при необходимости) набора семафоров
semop	Запрос и изменение значения семафора
semctl	Запрос и изменение управляющих параметров набора семафоров, удаление набора

Для этих API необходимы следующие файлы заголовков:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

10.5.3. Функция semget

Прототип функции *semget* имеет следующий вид:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key_t key, int num_sem, int flag );
```

Эта функция "открывает" набор семафоров, идентификатор которого задан значением аргумента *key*, и возвращает неотрицательный целочисленный дескриптор. Его можно использовать в других API семафоров для запроса и изменения значения семафора, а также для запроса и/или установки управляющих параметров набора семафоров.

Если значение аргумента *key* — положительное целое, рассматриваемый API пробует открыть набор семафоров, ключевой идентификатор которого совпадает с указанным значением. Если же значением *key* является макрос IPC_PRIVATE, API создает новый набор семафоров, который будет использоваться исключительно вызывающим процессом. Такие "частные" семафоры обычно выделяются родительским процессом, создающим затем один или несколько порожденных процессов. Родительский и порожденные процессы пользуются этими семафорами для синхронизации своей работы.

При нулевом значении аргумента *flag* API прерывает свою работу, если отсутствует набор семафоров, ключевой идентификатор которого совпадал бы с заданным значением *key*; в противном случае возвращается дескриптор этого набора. Если процессу необходимо создать новый набор с идентификатором *key* (и набора с таким идентификатором нет), то значение аргумента *flag* должно представлять собой результат побитового логического сложения константы IPC_CREAT и числовых значений прав доступа к новому набору для чтения и записи.

Значение *num_sem* может быть равно нулю, если флаг IPC_CREAT в аргументе *flag* не указан, или числу семафоров во вновь создаваемом наборе.

Например, следующий системный вызов создает двухсемафорный набор с идентификатором 15 и разрешением на чтение-запись для владельца, только на чтение для членов группы и прочих пользователей, если такого набора еще нет. Этот вызов возвращает целочисленный дескриптор, необходимый для последующих обращений к данному набору:

```
int perms = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
int semfdesc = semget(15, 2, IRC_CREAT | perms);
```

Для гарантированного создания нового набора семафоров можно указать одновременно с флагом IPC_CREAT флаг IPC_EXCL. Тогда API будет успешно выполнен только в том случае, если он создаст новый набор с заданным значением *key*.

В случае неудачи этот API возвращает -1.

10.5.4. Функция semop

Прототип функции *semop* имеет следующий вид:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (int semfd, struct sembuf* opPtr, int len );
```

С помощью этого API можно изменять значение одного или нескольких семафоров в наборе (указанном аргументом *semfd*) и/или проверять равенство их значений нулю. Аргумент *opPtr* — это указатель на массив объектов типа *struct sembuf*, каждый из которых задает одну операцию (запрос или изменение значения). Аргумент *len* показывает, сколько элементов имеется в массиве, указанном аргументом *opPtr*.

Тип данных *struct sembuf* определяется в заголовке <sys/sem.h>:

```
struct sembuf
{
    short  sem_num;   // индекс семафора
    short  sem_op;    // операция над семафором
    short  sem_flg;   // флаг(и) операции
};
```

Переменная *sem_op* может иметь следующие значения.

Значение *sem_op*	Смысл
Положительное число	Увеличить значение указанного семафора на эту величину
Отрицательное число	Уменьшить значение указанного семафора на эту величину
0	Проверить равенство значения семафора нулю

Если вызов *semop* попытается уменьшить значение семафора до отрицательного числа или посчитает, что значение семафора равно нулю, когда на самом деле это не так, то ядро заблокирует вызывающий процесс. Этого не произойдет в том случае, если в полях *sem_flg* элементов массива, где *sem_op* меньше или равно нулю, указан флаг IPC_NOWAIT.

В полях *sem_flg* объектов *struct sembuf* может быть установлен еще один флаг — SEM_UNDO. Этот флаг дает ядру указание отслеживать изменение значения семафора (произведенное вызовом *semop*). При завершении вызывающего процесса ядро ликвидирует сделанные изменения, чтобы процессы, ожидающие изменения семафоров, не были заблокированы навечно, что может произойти в том случае, если вызывающий процесс "забудет" отменить сделанные им изменения.

В случае успешного выполнения этот API возвращает нуль, а в случае неудачи — -1.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

/* уменьшить значение 1-го семафора на 1, проверить значение 2-го
   семафора на равенство нулю */
struct semid_ds sbuf[2] = {{0, -1, SEM_UNDO | IPC_NOWAIT}, {1, 0, 1}};

int main()
{
   int perms = S_IRWXU | S_IRWXG | S_IRWXO;
   int fd = semget(100, 2, IRC_CREAT | perms);
   if (fd==-1) { perror("semget"); exit; }
   if (semop(fd, sbuf, 2)==-1) perror("semop;");
   return 0;
}
```

В этом примере открывается двухсемафорный набор с ключевым идентификатором 100. Этот набор создается (если такого набора еще не было) с разрешением на чтение и запись для всех категорий пользователей.

Если вызов *semget* проходит успешно, процесс вызывает функцию *semop*, которая уменьшает значение первого семафора на 1 и проверяет значение второго семафора на равенство нулю. Кроме того, при обращении к первому семафору указываются флаги IPC_NOWAIT и SEM_UNDO.

10.5.5. Функция semctl

Прототип функции *semctl* имеет следующий вид:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (int semfd, int num, int cmd, union semun arg );
```

С помощью этого API можно запрашивать и изменять управляющие параметры набора семафоров, указанного аргументом *semfd*, а также удалять семафор.

Значение *semfd* — это дескриптор набора семафоров, который берется из вызова функции *semget*.

Значение *num* — это индекс семафора, а следующий аргумент, *cmd*, задает операцию, которая должна быть выполнена над конкретным семафором данного набора.

Аргумент *arg* — это объект типа *union*, который может использоваться для задания или выборки управляющих параметров одного или нескольких

семафоров набора в соответствии с аргументом *cmd*. Тип данных *union semun* определяется в заголовке <sys/sem.h>:

```
union semun
{
   int              val;     // значение семафора
   struct semid_ds  *buf;    // управляющие параметры набора
   ushort           *array;  // массив значений семафоров
};
```

Ниже перечислены возможные значения аргумента *cmd* и их смысл.

Значение *cmd*	Что должен сделать процесс
IPC_STAT	Копировать управляющие параметры набора в объект, указанный аргументом *arg.buf*. У вызывающего процесса должно быть право на чтение набора
IPC_SET	Заменить управляющие параметры набора семафоров данными, определенными в объекте, на который указывает аргумент *arg.buf*. Чтобы выполнить эту операцию, вызывающий процесс должен иметь права привилегированного пользователя, создателя или назначенного владельца набора. Этот API может устанавливать UID владельца набора и идентификатор его группы, а также права доступа
IPC_RMID	Удалить семафор из системы. Чтобы выполнить эту операцию, вызывающий процесс должен иметь права привилегированного пользователя, создателя или назначенного владельца набора
GETALL	Скопировать все значения семафоров в массив, на который указывает *arg.array*
SETALL	Установить все значения семафоров равными значениям, содержащимся в массиве, на который указывает *arg.array*
GETVAL	Возвратить значение семафора с номером *num*. Аргумент *arg* не используется
SETVAL	Установить значение семафора с номером *num* равным значению, указанному в *arg.val*
GETPID	Возвратить идентификатор процесса, который выполнял операции над семафором с номером *num* последним. Аргумент *arg* не используется
GETNCNT	Возвратить количество процессов, которые в текущий момент заблокированы и ожидают увеличения значения семафора с номером *num*. Аргумент *arg* не используется
GETZCNT	Возвратить количество процессов, которые в текущий момент заблокированы и ожидают обращения значения семафора с номером *num* в нуль. Аргумент *arg* не используется

В случае успешного выполнения этот API возвращает значение, соответствующее конкретному *cmd*, а в случае неудачи — -1.

Приведенная ниже программа *test_sem.C* "открывает" набор семафоров с ключевым идентификатором 100 и вызывает *semctl* для получения управляющих параметров набора. Если вызовы функций *semget* и *semctl* выполняются успешно, процесс направляет на стандартное устройство вывода количество семафоров, содержащихся в наборе. Посредством еще одного вызова *semctl*

процесс устанавливает UID владельца набора равным UID процесса. Наконец, вызвав *semctl* в третий раз, процесс удаляет набор.

```
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/sem.h>

union semun {
   int  val;
   struct semid_ds *mbuf;
   ushort *array;
} arg;

int main()
{
   struct semid_ds mbuf;
   arg.mbuf = &mbuf;
   int fd = semget (100, 0, 0);
   if (fd>0 && semctl(fd,0, IPC_STAT,arg)) {
    cout << "#semaphores in the set" << arg.mbuf->sem_nsems<< endl;
    arg.mbuf->sem_perm.uid = getuid(); // изменить UID владельца
    if (semctl(fd,0,IPC_SET,arg)==-1) perror("semctl");
   }
   else perror("semctl");
   if (semctl(fd,0,IPC_RMID,0)) perror("semctl - IPC_RMID");
   return 0;
}
```

10.6. Семафоры POSIX.1b

Семафоры POSIX.1b создаются и обрабатываются приблизительно так же, как семафоры UNIX System V. В частности, в стандарте POSIX.1b определяются заголовок <semaphore.h> и следующий набор API:

```
#include <semaphore.h>

sem_t sem_open ( char* name, int flags, mode_t mode, unsigned init_value );

int sem_init ( sem_t* addr, int pshared, unsigned init_value);
int sem_getvalue ( sem_t* idp, int* valuep );
int sem_close ( sem_t* idp );
int sem_destroy ( sem_t* id );
int sem_unlink ( char* name );
int sem_wait ( sem_t* idp );
int sem_trywait ( sem_t* idp );
int sem_post ( sem_t* idp );
```


Семафоры POSIX.1b отличаются от семафоров UNIX System V следующим:

- Семафоры POSIX.1b либо обозначаются путевым UNIX-именем (если создаются с помощью *sem_open*), либо остаются безымянными (но с присвоением начального виртуального адреса, если они создаются посредством вызова функции *sem_init*). Напомним, что семафоры System V обозначаются целочисленным ключом.
- В POSIX.1b с каждым вызовом *sem_open* и *sem_init* создается один семафор, тогда как каждый вызов *semget* в System V позволяет создать несколько семафоров.
- Значение семафора POSIX.1b увеличивается и уменьшается на единицу с каждым вызовом *sem_post* и *sem_wait* соответственно. В UNIX System V пользователи могут увеличивать и уменьшать значения семафоров на любую целую величину с каждым вызовом *semop*.

Функция *sem_open* создает семафор, имя которого задано аргументом *name*. Синтаксис аргумента *name* такой же, как в сообщениях POSIX.1b. Значение аргумента *flags* может быть равно нулю, если функции известно, что указанный семафор уже существует. Флаг O_CREAT показывает, что следует создать семафор с заданным именем. Вместе с этим флагом можно указывать флаг O_EXCL, который заставляет функцию возвращать код неудачного завершения, если семафор с заданным именем уже существует. Аргументы *mode* и *init_value* используются при создании нового семафора. Значение аргумента *mode* — это права доступа на чтение и запись для владельца, группы и прочих пользователей, которые должны быть установлены для нового семафора. Аргумент *init_value* задает значение, которое должно быть присвоено семафору. Тип этого аргумента — целое без знака.

В случае успешного выполнения рассматриваемая функция возвращает указатель на структуру типа *sem_t*, а в случае неудачи — -1.

Функция *sem_init* — альтернатива функции *sem_open*. Процесс, использующий *sem_init*, сначала выделяет область памяти для создаваемого семафора. Эта область памяти может быть разделяемой, если к семафору должны обращаться другие процессы. Адрес этой области передается как значение в аргумент *addr* функции *sem_init*. Аргумент *pshared* имеет значение 1, если семафор должен использоваться несколькими процессами совместно. В противном случае значение этого аргумента — 0. Аргумент *init_value* содержит целое значение, которое должно быть присвоено семафору как начальное. Это значение не должно быть отрицательным числом.

В случае успешного выполнения данная функция возвращает 0, а в случае неудачи — -1.

Функция *sem_getvalue* позволяет определить текущее значение семафора, указанного аргументом *idp*. Это значение передается с помощью аргумента *valuep*. В случае успешного выполнения данная функция возвращает 0, а в случае неудачи — -1.

Функция *sem_post* увеличивает значение семафора на единицу, а функция *sem_wait* уменьшает его на единицу. Семафор в обеих функциях обозначается

аргументом *idp*. Если его значение уже равно нулю, функция *sem_wait* блокирует вызывающий процесс до тех пор, пока не будут созданы условия для его успешного продолжения. Функция *sem_trywait* похожа на *sem_wait*, но она неблокирующая и возвращает -1 (если не может уменьшить указанное значение семафора).

Функции *sem_close* и *sem_unlink* используются с семафорами, созданными функцией *sem_open*. Первая из этих функций отсоединяет семафор от процесса, а вторая — удаляет его из системы.

Функция *sem_destroy* используется с семафорами, созданными функцией *sem_init*. Она удаляет семафор из системы.

В случае успешного выполнения функции *sem_post*, *sem_wait*, *sem_trywait*, *sem_close*, *sem_unlink* и *sem_destroy* возвращают 0, а в случае неудачи — -1.

В приведенной ниже программе *test_semp.C* создается набор семафоров с именем */sem.0*, который инициализируется значением 1. Если семафор создается нормально, процесс выполняет функцию *sem_wait*, которая уменьшает значение семафора до нуля, а затем функцию *sem_post*, которая вновь увеличивает его значение до единицы. Наконец, процесс закрывает указатель семафора с помощью функции *sem_close* и удаляет его из системы посредством вызова функции *sem_unlink*.

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

int main()
{
   sem_t *semp = sem_open("/sem.0", O_CREAT, S_IRWXU, 1);
   if (semp==(sem_t*)-1)            {perror("sem_open"); return -1;}
   if (sem_wait(semp)==-1)          perror("sem_wait");
   if (sem_post(semp)==-1)          perror("sem_post");
   if (sem_close(semp)==-1)         perror("sem_close");
   if (sem_unlink("/sem.0") == -1)  perror("sem_unlink");
   return 0;
}
```

В следующем примере, *test_semp2.C*, с помощью функции *sem_init* семафор размещается по адресу, полученному процессом с помощью функции *malloc*. Этот семафор задается как неразделяемый (*pshare = 0*), а его начальное значение устанавливается равным единице. Если он создается нормально, то вызывается API *sem_getvalue* для получения текущего значения и процесс направляет это значение на стандартное устройство вывода. Наконец, семафор удаляется из системы с помощью API *sem_destroy*.

```
#include <iostream.h>
#include <stdio.h>
#include <malloc.h>
#include <semaphore.h>

int main()
{
   int val;
   sem_t *semp = (sem_t*)malloc(sizeof(sem_t));
   if (!semp) { perror("malloc"); return 1; }
   if (sem_init (semp, 0, 1 )==-1) { perror("sem_init"); return 2; }
   if (sem_getvalue(semp, &val)==0)
     cout << "semaphore value: " << val << endl;
   if (sem_destroy(semp) == -1) perror("sem_destroy");
   return 0;
}
```

10.7. Разделяемая память в UNIX System V

Разделяемая память позволяет множеству процессов отображать часть своих виртуальных адресов в общую область памяти. Благодаря этому любой процесс может записывать данные в разделяемую область памяти, и эти данные будут доступны для чтения и модификации другим процессам.

Разделяемая память была задумана как средство повышения производительности при обработке сообщений. Дело в том, что при передаче сообщения из процесса в очередь сообщений данные копируются из виртуального адресного пространства процесса в область данных ядра. Затем, когда другой процесс принимает это сообщение, ядро копирует данные сообщения из своей области в виртуальное адресное пространство процесса-получателя. Таким образом, содержащиеся в сообщении данные копируются дважды: вначале из процесса в ядро, а затем в другой процесс. При наличии разделяемой памяти такие дополнительные издержки отсутствуют: эта память выделяется в виртуальном адресном пространстве ядра всякий раз, когда процессу необходимо записывать данные. Они обрабатываются непосредственно в области памяти ядра. При этом, однако, в разделяемой памяти не предусмотрены какие-либо методы управления доступом для процессов, которые ею пользуются. Поэтому среда межпроцессного взаимодействия формируется, как правило, путем использования разделяемой памяти и семафоров.

Процесс, подсоединившийся к разделяемой области памяти, получает указатель на эту разделяемую память. Им можно пользоваться так, как будто он получен в результате вызова функции динамического распределения памяти (т.е. с помощью функции *new*). Единственное различие состоит в том, что данные в разделяемой памяти не исчезают, даже если процесс, создавший разделяемую область, завершается.

В любой момент времени в системе может существовать множество разделяемых областей памяти.

10.7.1. Поддержка разделяемой памяти ядром UNIX

В UNIX System V.3 и V.4 в адресном пространстве ядра имеется таблица разделяемой памяти, в которой отслеживаются все разделяемые области

памяти, создаваемые в системе. В каждом элементе таблицы находятся следующие данные об одной разделяемой области памяти:

- Имя, представляющее собой целочисленный идентификационный ключ, присвоенный разделяемой области памяти процессом, который ее создал. Другие процессы могут, указывая этот ключ, "открывать" данную область и получать дескриптор для дальнейшего обращения к области и отсоединения от нее.
- UID создателя разделяемой области памяти и идентификатор его группы. Процесс, эффективный UID которого совпадает с UID создателя разделяемой области памяти, может удалять область и изменять параметры управления ею.
- UID назначенного пользователя-владельца и идентификатор его группы. Эти идентификаторы обычно совпадают с идентификаторами создателя, но создатель может устанавливать эти идентификаторы для переназначения владельца области и принадлежности к группе.
- Права доступа к области для чтения-записи по категориям "владелец", "группа" и "прочие". Процесс, имеющий право на чтение из данной области, может читать из нее данные и запрашивать UID назначенного пользователя и его группы. Процесс, имеющий разрешение на запись в область, может записывать в нее данные.
- Размер разделяемой области памяти в байтах.
- Время, когда какой-либо процесс в последний раз подсоединялся к области.
- Время, когда какой-либо процесс в последний раз отсоединялся от области.
- Время, когда какой-либо процесс в последний раз изменил управляющие параметры области.

На рис. 10.5 изображена структура данных ядра, предназначенная для манипулирования разделяемой памятью.

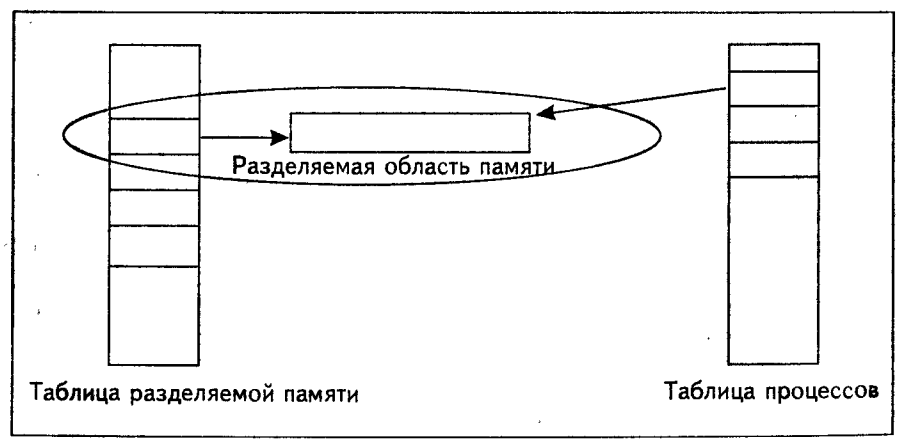


Рис. 10.5. Структура данных ядра для манипулирования разделяемой памятью

Аналогично сообщениям и семафорам, разделяемая память располагается в адресном пространстве ядра, и разделяемые области не освобождаются, даже если создавшие их процессы завершаются. У каждой разделяемой области памяти есть назначенный владелец, и удалять эту область или корректировать ее управляющие параметры могут только процессы с правами привилегированного пользователя, создателя или назначенного владельца. Для манипулирования разделяемой памятью система устанавливает ряд ограничений. Они определяются в заголовке <sys/shm.h>:

Системное ограничение	Значение
SHMMNI	Максимальное число разделяемых областей памяти, которые могут существовать в системе в любой данный момент времени
SHMMIN	Минимальный размер разделяемой области памяти в байтах
SHMMAX	Максимальный размер разделяемой области памяти в байтах

Влияние этих системных ограничений на процессы таково:

- если попытка процесса создать новую разделяемую область памяти вызовет превышение лимита SHMMNI, этот процесс будет блокироваться до тех пор, пока другой процесс не удалит одну из существующих областей;
- если процесс попытается создать область с размером меньше SHMMIN или больше SHMMAX, этот системный вызов выполнен не будет.

10.7.2. API ОС UNIX для разделяемой памяти

В заголовке <sys/ipc.h> объявляется тип данных *struct ipc_perm*, используемый для хранения UID создателя, UID владельца, идентификаторов их групп, присвоенного ключевого имени и прав на чтение и запись, установленных для разделяемой области памяти.

Элементы таблицы разделяемой памяти относятся к типу данных *struct shmid_ds*, который определяется в заголовке <sys/shm.h>. Ниже перечислены информационные поля этой структуры и данные, которые в них хранятся.

Поле	Данные
shm_perm	Данные, хранящиеся в записи *struct ipc_perm*
shm_segsz	Размер разделяемой области памяти в байтах
shm_lpid	Идентификатор процесса, который в последний раз подсоединялся к области
shm_cpid	Идентификатор процесса-создателя
shm_nattch	Число процессов, подсоединенных к области в данный момент
shm_atime	Время, когда процесс в последний раз подсоединялся к области

Поле	Данные
shm_dtime	Время, когда процесс в последний раз отсоединялся от области
shm_ctime	Время, когда последний процесс изменил управляющие параметры области

Для манипулирования разделяемой областью памяти в UNIX System V имеются четыре API:

API разделяемой памяти	Назначение
shmget	Создание и открытие разделяемой области памяти
shmat	Подсоединение разделяемой области памяти к виртуальному адресному пространству процесса с тем, чтобы этот процесс мог читать данные из разделяемой памяти и/или записывать в нее данные
shmdt	Отсоединение разделяемой области памяти от виртуального адресного пространства процесса
shmctl	Запрос и изменение управляющих параметров разделяемой области памяти, а также ее удаление

Для перечисленных в таблице API разделяемой памяти необходимы следующие файлы заголовков:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

10.7.3. Функция shmget

Прототип функции *shmget* имеет следующий вид:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key_t key, int size, int flag );
```

Этот API открывает разделяемую область памяти, идентификатор которой совпадает со значением аргумента *key*, и возвращает неотрицательный целочисленный дескриптор. Его можно использовать в других API разделяемой памяти.

Если значение аргумента *key* — положительное целое, данный API пробует открыть разделяемую область памяти, ключевой идентификатор которой совпадает с этим значением. Если же значением *key* является макрос IPC_PRIVATE, API выделяет новую разделяемую область памяти, которая будет использоваться исключительно вызывающим процессом. Такая "частная" разделяемая область памяти обычно выделяется родительским процессом,

который затем создает один или несколько порожденных процессов. Родительский и порожденные процессы пользуются этой разделяемой памятью для обмена данными.

Аргумент *size* задает размер области разделяемой памяти, которая может быть подсоединена к вызывающему процессу с помощью API *shmat*. Если в результате этого вызова создается новая область разделяемой памяти, ее размер будет определяться аргументом *size*. В случае, когда вызов открывает уже существующую область, значение аргумента *size* может быть меньше размера выделенной области разделяемой памяти или равно ему. Если в последнем случае значение *size* меньше фактического размера области разделяемой памяти, то вызывающий процесс может получить доступ только к первым *size* байтам области разделяемой памяти.

Если аргумент *flag* равен нулю и нет области разделяемой памяти, идентификатор которой совпадал бы с заданным значением *key*, то этот API завершается неудачно. В противном случае он возвращает дескриптор этой области. Если процессу необходимо создать разделяемую область памяти с заданным *key* (и области с таким идентификатором нет), то значение аргумента *flag* должно представлять собой результат побитового логического сложения константы IPC_CREAT и прав доступа к новой области памяти для чтения и записи.

В случае успешного выполнения эта функция возвращает положительный дескриптор, а в случае неудачи — -1.

10.7.4. Функция shmat

Прототип функции *shmat* выглядит следующим образом:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void* shmat ( int shmid, void* addr, int flag );
```

Эта функция подсоединяет область разделяемой памяти, указанную аргументом *shmid*, к виртуальному адресному пространству вызывающего процесса. Процесс может затем читать данные из указанной области памяти и записывать в нее данные. Если это вновь создаваемая область разделяемой памяти, то ядро реально выделяет область памяти только тогда, когда первый процесс вызывает рассматриваемую функцию для подсоединения к новой области.

Аргумент *addr* задает начальный виртуальный адрес адресного пространства вызывающего процесса, в которое необходимо отобразить разделяемую память. Если это значение равно нулю, ядро может само найти в вызывающем процессе подходящий виртуальный адрес для отображения разделяемой памяти. Большинство приложений должны устанавливать значение *addr* в

нуль, если они явно не хранят в разделяемой области памяти ссылки на указатели или адреса (например, не держат в этой области связный список). Очень важно, чтобы каждый процесс, обращающийся к данной области памяти, указывал один и тот же начальный виртуальный адрес области, в которую отображена разделяемая память.

Если значение аргумента *addr* не равно нулю, аргумент *flag* может содержать флаг SHM_RND. Этот флаг указывает ядру на то, что виртуальный адрес, заданный в аргументе *addr*, можно округлить до границы страницы памяти. Если флаг SHM_RND отсутствует и значение аргумента *addr* не равно нулю, соответствующий API завершается неудачно (это означает, что ядро не может отобразить разделяемую память в область, заданную виртуальным адресом).

Аргумент *flag* может иметь также значение SHM_RDONLY, которое говорит о том, что вызывающий процесс подсоединяется к разделяемой памяти только для чтения. Если этот флаг не установлен, то по умолчанию процесс может читать и записывать данные в разделяемую память с учетом разрешений, установленных создателем данной области.

Этот API возвращает виртуальный адрес области отображения разделяемой памяти, а в случае неудачи — -1. Следует отметить, что любой процесс, с целью подсоединения одной разделяемой области памяти к виртуальным адресным пространствам многих процессов, может вызывать функцию *shmat* многократно.

10.7.5. Функция shmdt

Прототип функции *shmdt* имеет следующий вид:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt ( void* addr );
```

Эта функция отсоединяет разделяемую память от заданного аргументом *addr* виртуального адреса вызывающего процесса.

Прежде чем вызывать данную функцию, необходимо получить посредством вызова *shmat* значение *addr*.

В случае успешного выполнения рассматриваемая функция возвращает 0, а в случае неудачи — -1.

Приведенная ниже программа *test_shm.C* открывает разделяемую область памяти размером 1024 байта с ключевым идентификатором 100. Если такая область памяти не существует, то программа создает ее с помощью вызова *shmget* и устанавливает для нее разрешение на чтение-запись для всех пользователей.

После открытия разделяемая память подсоединяется к виртуальному адресу процесса посредством вызова функции *shmat*. Затем программа записывает сообщение *Hello* в разделяемую область памяти и отсоединяет от нее процесс. После этого любой процесс может подсоединяться к разделяемой области и читать записанное в ней сообщение.

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
    int perms = S_IRWXU | S_IRWXG | S_IRWXO;
    int fd = shmget (100, 1024, IPC_CREAT | perms);
    if (fd==-1) perror("shmget"), exit(1);
    char* addr = (char*)shmat(fd, 0, 0);
    if (addr==(char*)-1) perror("shmat"), exit(1);
    strcpy( addr, "Hello");
    if (shmdt(addr)==-1) perror("shmdt");
    return 0;
}
```

10.7.6. Функция semctl

Прототип функции *semctl* имеет следующий вид:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (int shmid, int cmd, struct shmid_ds* buf );
```

С помощью этого API можно запрашивать управляющие параметры разделяемой области памяти, указанной аргументом *shmid*, изменять эти параметры и удалять данную область памяти.

Значение *shmid* — это дескриптор разделяемой области памяти, полученный посредством вызова функции *shmget*.

Аргумент *buf* — это адрес объекта типа *struct shmid_ds*, который можно использовать для задания и выборки управляющих параметров разделяемой памяти, указанных аргументом *cmd*. Ниже перечислены возможные значения аргумента *cmd* и вызываемые ими действия:

Значение *cmd*	Что должен сделать процесс
IPC_STAT	Копировать управляющие параметры разделяемой области памяти в объект, указанный аргументом *buf*
IPC_SET	Заменить управляющие параметры разделяемой области памяти параметрами, определенными в объекте, на который указывает аргумент *buf*. Чтобы выполнить эту операцию, вызывающий процесс должен иметь права привилегированного пользователя, создателя или назначенного владельца разделяемой памяти. Рассматриваемый API может устанавливать только UID владельца области и идентификатор его группы, а также права доступа
IPC_RMID	Удалить разделяемую область памяти из системы. Чтобы выполнить эту операцию, вызывающий процесс должен иметь права привилегированного пользователя, создателя или назначенного владельца области. Если к разделяемой области памяти, подлежащей удалению, подсоединены один или несколько процессов, то операция удаления будет отложена до тех пор, пока эти процессы не отсоединятся от нее
SHM_LOCK	Блокировать разделяемую область памяти. Для выполнения этой операции вызывающий процесс должен обладать правами привилегированного пользователя
SHM_UNLOCK	Разблокировать разделяемую область памяти. Для выполнения этой операции вызывающий процесс должен обладать правами привилегированного пользователя

В случае успешного выполнения рассматриваемый API возвращает 0, а в случае неудачи — -1.

Приведенная ниже программа *test_shm2.C* открывает разделяемую память с ключевым идентификатором 100 и вызывает API *shmctl* для выборки управляющих параметров этой области. Если вызовы *shmget* и *shmctl* выполняются успешно, процесс выводит на экран размер разделяемой области. Посредством еще одного вызова *shmctl* процесс устанавливает идентификатор владельца очереди равным своему UID. Наконец, вызвав *shmctl* в третий раз, процесс удаляет разделяемую область памяти.

```
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
   struct shmid_ds sbuf;
   int fd = shmget (100, 1024, 0);
   if (fd>0 && shmctl(fd, IPC_STAT,&sbuf)) {
     cout << "shared memory size if: " << sbuf.shm_segsz << endl;
      sbuf.shm_perm.uid = getuid();     // заменить UID владельца
     if (shmctl(fd,IPC_SET,&sbuf)==-1) perror("shmctl");
   }
   else perror("shmctl");
   if (shmctl(fd,IPC_RMID,0)) perror("shmctl - IPC_RMID");
   return 0;
}
```


10.7.7. Пример приложения клиент/сервер с семафорами и разделяемой памятью

В данном разделе описывается еще одна версия приложения клиент/сервер, которое мы рассматривали в разделе 10.3.7. Вместо сообщений здесь для организации очереди сообщений используются семафоры и разделяемая память. По этой причине заголовок *message.h* требует серьезной корректировки, однако интерфейс класса *message* клиентской и серверной программ, а также модули *client.C* и *server.C* — те же самые, которые описывались в разделе 10.3.7. В этом состоит преимущество применения классов C++: если внешние интерфейсы класса не меняются, то не нужно модифицировать ни одно приложение, которое использует этот класс, даже если внутренняя реализация класса существенно изменилась.

Для создания очереди сообщений при помощи семафоров и разделяемой памяти в адресном пространстве ядра создается область памяти, совместно используемая для записи всех сообщений, передаваемых в эту очередь. Семафоры определяют, какой процесс может получить доступ к этой разделяемой области памяти (для чтения и записи сообщения) в каждый момент времени. В частности, может выполняться множество клиентских процессов, одновременно передающих запросы в серверный процесс и манипулирующих семафорами посредством одних и тех же системных вызовов *semop*. Обработка этих вызовов должна обеспечивать один из двух вариантов: либо все они блокируются, когда сервер активно работает с разделяемой памятью, либо только один из клиентских процессов активно работает с этой областью памяти (и тогда все остальные клиентские процессы и серверный процесс блокируются). Чтобы реализовать эти варианты, используются два семафора со следующими возможными значениями:

Семафор 0	Семафор 1	Значение
0	1	Сервер ожидает передачи сообщения клиентом
1	0	Сообщение клиента готово для прочтения сервером
1	1	Данные ответа сервера клиенту готовы

Клиентские процессы и серверный процесс взаимодействуют с этим набором семафоров следующим образом.

- Сначала сервер создает набор семафоров и разделяемую область памяти. Он инициализирует созданный набор значениями 0, 1 (т.е. значение первого семафора — 0, а значение второго семафора — 1).
- Сервер ждет, когда клиент передаст запрос в разделяемую область памяти. Для этого он выполняет вызов *semop* с заданными значениями -1 и 0. Такой вызов блокирует сервер, потому что текущее значение набора — 0, 1 и ни один из семафоров набора не может быть изменен вызовом *semop* со значениями -1, 0.

- Когда один или несколько клиентов пытаются передать сообщение в разделяемую область памяти, они выполняют вызов *semop* с заданными значениями 0, -1. Один из этих вызовов завершится успешно, потому что на тот момент значение семафора будет равно 0, 1. Клиентский процесс, успешно выполнив вызов *semop*, сразу же изменит значения семафоров на 0, 0. В результате будут блокированы все остальные клиентские процессы, которые выполняют вызов *semop* со значением 0, -1, а также сервер, выполняющий вызов *semop* со значением -1, 0.
- Клиентский процесс, изменивший значения набора семафоров, может теперь записать в разделяемую область памяти команду запроса на обслуживание и свой идентификатор. После этого он выполняет вызов *semop* со значением 1, 0. В результате серверный процесс разблокируется и может вызвать функцию *semop*, но новые значения семафоров будут по-прежнему блокировать остальные клиентские процессы, которые выполняют вызов *semop* со значением 0, -1. Если команда запроса на обслуживание — не QUIT_CMD, этот клиентский процесс выполнит вызов *semop* со значениями -1, -1 и заблокируется.
- Разблокированный сервер прочитает из разделяемой области памяти запрос на обслуживание, записанный клиентом. Если была записана команда QUIT_CMD, сервер освободит разделяемую область памяти и набор семафоров, а затем завершит свою работу. Если записанная команда — не QUIT_CMD, сервер запишет данные ответа в разделяемую область памяти, а затем выполнит вызов semop со значением 1, 1. Это разблокирует клиентский процесс, который выполняет вызов *semop* со значением -1, -1. Остальные клиентские процессы, которые выполняют вызов *semop* со значением 0, -1, остаются заблокированными новыми значениями семафоров. После вызова *semop* сервер возвращается в состояние ожидания запроса на обслуживание от нового клиента.
- Клиент, который разблокирован сервером, устанавливает значение набора семафоров в 0, 0 и читает данные ответа сервера. Он посылает данные на стандартное устройство вывода, а затем устанавливает значения семафоров в 0, 1, после чего завершается. Последний вызов *semop* возвращает систему в состояние, в котором один из клиентов будет разблокирован и начнет взаимодействовать с сервером через разделяемую область памяти и семафоры.

Последовательность изменения значений семафоров на различных этапах взаимодействия между клиентом и сервером представлена на рис. 10.6 (значения семафоров показаны в овалах).

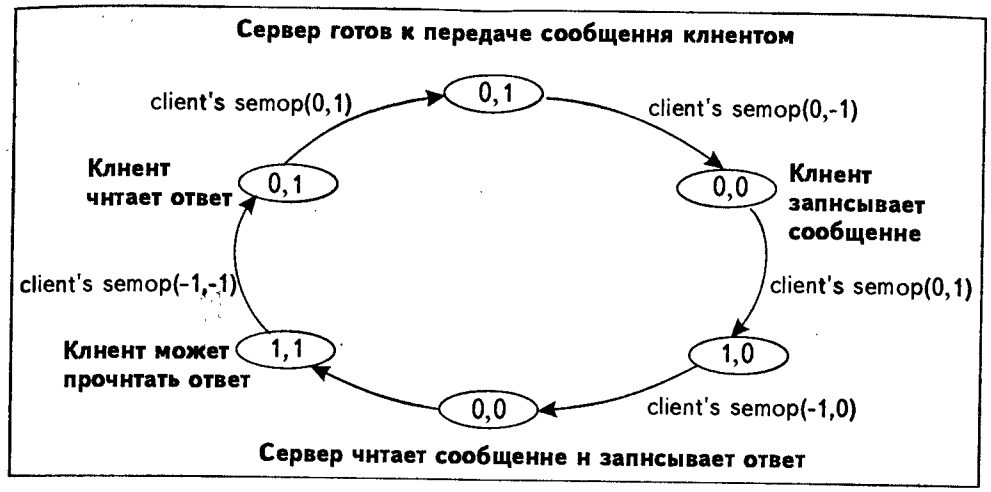


Рис. 10.6. Взаимодействие между клиентом и сервером с использованием семафоров и разделяемой области памяти

Заголовок *message.h*, описанный в разделе 10.3.7, модифицируется под использование разделяемой области памяти и семафоров. Новый класс *message* объявляется в следующем заголовке *message3.h*:

```
#ifndef MESSAGE3_H
#define MESSAGE3_H
#include <strstream.h>
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <sys/errno.h>

/* общие объявления для серверного процесса и демона */
enum { MSGKEY=186, MAX_LEN=256, SHMSIZE=1024, SEMSIZE=2 };
enum { LOCAL_TIME = 1, UTC_TIME = 2, QUIT_CMD = 3, ILLEGAL_CMD = 4,
       SEM_RD = 0, SEM_WR=1 };

struct mgbuf
{
   long    mtype;
   char     mtext[MAX_LEN];
};

class message
{
```

```
private:
  int    shmId, semId;
  struct mgbuf  *msgPtr;
  enum ipc_op ( RESET_SEM, CLIENT_GET_MEM, CLIENT_SND_REQ,
                SERVER_RCV_REQ, SERVER_GET_MEM, SERVER_SND_RPY,
                CLIENT_RCV_RPY};
public:
  /* попробуем изменить значения семафоров */
  void getsem( enum ipc_op  opType )
  {
     static struct sembuf  args[2] = { {SEM_RD}, {SEM_WR} };
     switch (opType) {
        case SERVER_GET_MEM:
           return;
        case CLIENT_GET_MEM:
           args[SEM_RD].sem_op = 0,
           args[SEM_WR].sem_op = -1;
           break;
        case CLIENT_SND_REQ:
           args[SEM_RD].sem_op = 1,
           args[SEM_WR].sem_op = 0;
           break;
        case SERVER_RCV_REQ:
           args[SEM_RD].sem_op = -1,
           args[SEM_WR].sem_op = 0;
           break;
        case SERVER_SND_RPY:
           args[SEM_RD].sem_op = 1,
           args[SEM_WR].sem_op = 1;
           break;
        case CLIENT_RCV_RPY:
           args[SEM_RD].sem_op = -1,
           args[SEM_WR].sem_op = -1;
           break;
        case RESET_SEM:
           args[SEM_RD].sem_op = 0,
           args[SEM_WR].sem_op = 1;
     }
     if (semop(semId,args,SEMSIZE)==-1) perror("semop");
  };

  /* функция-конструктор */
  message( int key )
  {
     if ((shmId=shmget(key, SHMSIZE, 0))==-1) {
       if (errno==ENOENT) { // создать новый объект класса message
          if ((shmId= shmget(key, SHMSIZE, IPC_CREAT|0666)) ==-1)
              perror("shmget");
          else if ((semId=semget(key, SEMSIZE, IPC_CREAT|0666))==-1)
              perror("semget");
          else getsem(RESET_SEM);  // инициализировать новый
                                   // набор семафоров
```



```
          }
          else perror("shmget");
       }
       else if ((semId=semget(key,0,0))==-1)  /* получить значения
                                                 существующих
                                                 семафоров */
          perror("semget");
       if (shmId>=0 && !(msgPtr=(struct mgbuf*)shmat(shmId,0,0)))
          perror("shmat");
     };

     /* функция-деструктор */
     ~message() {};

     /* проверить статус открытия очереди сообщений */
     int good() { return (shmId >= 0 && semId>=0) ? 1 : 0; };

      /* удалить очередь сообщений */
      int rmQ()
      {
        if (shmdt((char*)msgPtr)<0) perror("semdt");
        if (!semctl(semId,0,IPC_RMID,0) && !shmctl(shmId,IPC_RMID,0))
           return 0;
        perror("shmctl or semctl");
           return -1;
     };

     /* передать сообщение */
     int send( const void* buf, int size, int type)
     {
         int server = (type > 99);
         getsem(server ? SERVER_GET_MEM : CLIENT_GET_MEM);
         memcpy(msgPtr->mtext,buf,size);
         msgPtr->mtext[size] = '\0';
         msgPtr->mtype = type;
         getsem(server ? SERVER_SND_RPY : CLIENT_SND_REQ);
         return 0;
     };
     /* принять сообщение */
     int rcv( void* buf, int size, int type, int* rtype)
     {
         int server = (type < 0);
         getsem(server ? SERVER_RCV_REQ : CLIENT_RCV_RPY);
         memcpy(buf,msgPtr->mtext,strlen(msgPtr->mtext)+1);
         if (rtype) *rtype = msgPtr->mtype;
         if (!server) getsem(RESET_SEM);
         return strlen(msgPtr->mtext);
     };
};   /* message */
#endif
```

Утилита *getsem*, определенная в новом заголовке *message.h*, выполняет вызов *semop* с набором семафоров на основании фактических значений аргумента *opType* (они присваиваются либо серверным, либо клиентским процессом).

Функция-конструктор *message* открывает разделяемую область памяти и двухсемафорный набор. Эти два объекта имеют одинаковые ключевые идентификаторы. Если набор семафоров — совершенно новый объект, он инициализируется начальными значениями 0, 1 (т.е. значение первого семафора — 0, а второго — 1).

Функция *send* передает сообщение в разделяемую область памяти. Поскольку операции, выполняемые функцией *semop*, в серверном и клиентском процессах разные, функция *send* используется с аргументом *type*, который позволяет ей определить, каким является вызывающий процесс — серверным или клиентским. Если значение аргумента *type* превышает 99, функцию вызывает клиентский процесс. В противном случае это серверный процесс. Данная функция блокирует вызов *getsem* до тех пор, пока процесс не сможет завершить свою операцию *semop*. После этого процесс записывает сообщение в разделяемую область памяти и вновь вызывает *getsem* для установки значений семафоров, которые разблокируют клиентский процесс.

Функция *read* действует аналогичным образом. Операции *semop*, с помощью которых обеспечивается чтение сообщений из разделяемой области памяти, в серверном и клиентском процессах выполняются по-разному. Чтобы определить, каким является вызывающий процесс, функция *read* использует аргумент *type*. Если значение этого аргумента меньше нуля (реально это -99), вызывающий процесс — серверный; в противном случае — клиентский. Данная функция блокирует вызов *getsem* до тех пор, пока процесс не сможет завершить свою операцию *semop*. После этого процесс читает сообщение из разделяемой области памяти и сбрасывает семафоры в начальные значения 0, 1 (если процесс — клиентский).

Функция *rmQ* вызывается в том случае, если серверный процесс получает от клиентского процесса команду QUIT_CMD. Данная функция вызывает API *semctl* и *shmctl* для удаления набора семафоров и разделяемой области памяти, а затем для завершения серверного процесса. Это необходимо потому, что семафоры и разделяемая область памяти — устойчивые объекты, т.е. они сохраняются в адресном пространстве ядра даже после завершения процессов, которые их создали.

Результат выполнения клиентского и серверного процессов, использующих новый заголовок *message*.h, похож на результат, полученный при использовании сообщений (см. раздел 10.3.7):

```
chp13 % mserver &
[1] 356
server: start execution...
chp13 % mclient
client: start execution...
cmd> 1
```



```
server: receive cmd #1, from client: 338
client: 338 Tue Jan 26 21:50:59 1995
cmd> 2
server: receive cmd #2, from client: 338
client: 338 Fri Jan 27 05:51:19 1995
cmd> 4
server: receive cmd #4, from client: 338
client: 338 Illegal cmd: 4
cmd> 3
client: 338 exiting...
server: receive cmd #3, from client: 338
client: deleting msg queue...
[1] Done    mserver
chp13%
```

10.8. Ввод-вывод с отображением в память

API *mmap* был впервые предложен в BSD UNIX. Он позволяет процессу отображать свое виртуальное адресное пространство непосредственно на страницу памяти файлового объекта, находящуюся в пространстве ядра. Такой процесс может записывать данные файла непосредственно в эту отображенную область памяти и читать их из нее. Более того, если отображение в один файловый объект одновременно осуществляют несколько процессов, то они совместно используют отображенную область памяти. Эти процессы могут взаимодействовать друг с другом приблизительно так же, как при использовании разделяемой памяти.

API *mmap* отличается от обычных файловых API ОС UNIX тем, что если файл открывается с помощью вызова *read* (для чтения данных из файла), то ядро выбирает одну или несколько страниц затребованных данных из области на жестком диске, в которой хранится файл. Эти данные затем помещаются в отображенную область памяти ядра и копируются в буфер, находящийся в виртуальном адресном пространстве вызывающего процесса. В API *write* происходит обратное: когда процесс вызывает *write* для записи данных в файл, ядро копирует эти данные из буфера процесса в свою область памяти. Когда эта область заполняется или если процесс требует освободить буфер, данные копируются в файл, находящийся на жестком диске.

Если этот же процесс использует API *mmap*, ядро также выбирает одну или несколько страниц данных из файла на жестком диске и помещает их в область памяти ядра. В этом случае процесс может непосредственно обращаться к данным, находящимся в области памяти ядра, по виртуальным адресам, отображенным на эту область. Это делает *mmap* более эффективным инструментом манипулирования данными, чем обычные файловые API. Все данные, записываемые в отображенную область памяти, автоматически сохраняются в соответствующем файле.

Одно из применений *mmap* — разработка программ, которые могут возобновлять свое выполнение после остановки. Например, программа управления базой данных может с помощью *mmap* отобразить свое виртуальное

адресное пространство в файл базы данных, и все данные, которыми она манипулирует, будут сохранены в этой отображенной области. При завершении процесса эти данные автоматически сохраняются в файле базы данных. Когда программа возобновляет свое выполнение, новый процесс отображает часть своего виртуального адресного пространства в файл базы данных и все ранее записанные данные становятся доступными для дальнейшего использования.

Еще один вариант применения API *mmap* — эмуляция функции разделяемой памяти. Два и более процессов, которые должны взаимодействовать между собой, могут с помощью *mmap* осуществить отображение в один файловый объект. После этого они могут читать и записывать данные друг друга через свои отображенные виртуальные адреса. Ниже мы покажем, как с помощью *mmap* можно реализовать приложение клиент/сервер, рассмотренное в предыдущем разделе.

10.8.1. API ввода-вывода с отображением в память

Все API семейства *mmap* объявляются в заголовке <sys/mman.h>.

API	Назначение
mmap	Отображает виртуальное адресное пространство процесса в адресное пространство файлового объекта
munmap	Отсоединяет виртуальный адрес процесса от файлового объекта
msync	Синхронизирует данные в отображенной области памяти с данными соответствующего файлового объекта, хранящегося на жестком диске

10.8.2. Функция mmap

Прототип функции *mmap* имеет следующий вид:

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

caddr_t mmap ( caddr_t addr, int size, int prot, int flags, int fd, off_t pos );
```

Эта функция отображает файловый объект, указанный аргументом *fd*, в виртуальное адресное пространство процесса, начиная с адреса *addr*. Если аргумент *addr* равен нулю, ядро само назначает виртуальный адрес для отображения. Аргумент *pos* задает начальную позицию в файловом объекте, который отображается в *addr*. Его значение должно быть либо равным нулю, либо кратным размеру страницы памяти (для получения значения размера страницы системной памяти можно воспользоваться API *getpagesize* или *sysconf*). Аргумент *prot* задает права доступа к отображенной памяти. Он может иметь одно из следующих значений:

Значение *prot*	Смысл
PROT_READ	Данные, находящиеся в отображенной области памяти, можно читать
PROT_WRITE	В отображенную область памяти можно записывать
PROT_EXEC	Содержимое отображенной области памяти — исполняемый код

Аргумент *flags* задает опции отображения. Он может иметь одно из следующих значений:

Значение *flags*	Смысл
MAP_SHARED	Данные, записанные в отображенную область памяти, доступны другим процессам, которые отображаются в этот же файловый объект
MAP_PRIVATE	Данные, записанные в отображенную область, недоступны другим процессам, которые отображаются в этот же файловый объект
MAP_FIXED	Значение *addr* должно быть начальным виртуальным адресом отображенной области. Если отображение не может быть выполнено, функция возвращает код неудачного завершения. Если данный флаг не указан или если флаг MAP_VARIABLE определен и задан системой, то ядро может вместо *addr* выбрать для отображаемой области другой виртуальный адрес

Рассматриваемая функция возвращает фактический виртуальный адрес, с которого начинается отображенная область памяти. В случае неудачи возвращается MAP_FAILED.

Флаг PROT_EXEC используется в том случае, если отображаемый файловый объект — исполняемый файл и вызывающий процесс имеет права привилегированного пользователя. Когда пользователь вызывает команду в UNIX-системе, ядро обычно выполняет функцию *mmap* с исполняемым файлом этой команды. Содержимое исполняемого файла перед выполнением считывается непосредственно из области памяти, в которой отображен данный файл.

Флаг MAP_PRIVATE указывает на то, что все данные, записываемые в отображенную область памяти, невидимы для остальных процессов, отображенных в адресное пространство этого же файлового объекта. При этом, однако, названный флаг запрещает также и обратное сохранение данных (т.е. сохранение данных, записанных в отображенной области памяти, в файле, находящемся на диске). Предположим, что процессы А и Б отображаются в файл FOO, при этом процесс А установил флаг MAP_PRIVATE, а процесс Б — флаг MAP_SHARED. Если процесс Б запишет данные в отображенную область памяти раньше, чем это сделает процесс А, новые данные будут видимы обоим процессам. Когда процесс А записывает данные в совместно используемую отображенную область памяти, ядро создает отдельную копию страниц памяти, которые этот процесс изменил, тогда как процесс Б продолжает работать со старой страницей. С этого момента все данные, которые процесс А записывает на свою страницу памяти, перестают быть видимыми для процесса Б, и наоборот.

Процесс-отладчик использует флаг MAP_PRIVATE для отображения программы в память с целью ее последующего исполнения. Отладчик часто включает в код отлаживаемой программы определяемые пользователем контрольные точки. Эти точки не должны появиться в исполняемом файле, находящемся на диске, и не должны быть видимы другим процессам, которые отображаются в адресное пространство этой же программы.

Перед тем как вызывать функцию *mmap* для отображения файла, процесс должен вызвать API *open* и присвоить аргументу *fd* дескриптор открываемого файла. Кроме того, если этот файловый объект создается вызовом *open* заново, процесс должен записать в него для инициализации как минимум *size* байтов. Это делается потому, что *mmap* не выделяет память, а просто отмечает, что виртуальное адресное пространство процесса (от *addr* до *addr+size*) разрешается использовать. Если размер файла меньше, то ядро не выделяет память за пределами страницы, содержащей виртуальный адрес *addr+<file_size>*. Процесс, обращающийся к данным, расположенным между *addr+<file_size>* и *addr+size*, может получить сигнал SIGBUS.

Приведенная ниже программа *test_mmap.C* открывает новый файл с именем FOO размером SHMSIZE байтов (инициализированных значением "\0"), а затем закрывает дескриптор файла *fd*, потому что он больше не нужен. Наконец, процесс записывает в отображенную область памяти строку *Hello* и завершается. Если пользователь просмотрит содержимое этого файла, воспользовавшись для этого командой *cat*, он заметит, что файл содержит строку *Hello*.

```
#include <strstream.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
const int SHMSIZE = 1024;
#ifndef MMAP_FAILED
#define MMPAP_FAILED  (caddr_t*)0
#endif

int main()
{
    int ch='\0', fd = open ("FOO", O_CREAT|O_EXCL,0666);
    if (fd==-1) { perror("file exists"), exit(1); };
    for (int i=0; i < SHMSIZE; i++)        /* инициализировать файл */
      write(fd, &ch, 1);
    caddr_t memP=mmap(0,SHMSIZE,PROT_READ|PROT_WRITE, MAP_SHARED,fd, 0);
    if (memP==MAP_FAILED) {
      perror("mmap");
      exit(2);
    }
    close(fd);                          /* этот файл больше не нужен */
     ostrstream(memP,SHMSIZE) << "Hello UNIX\n";
    return 0;
}
```


10.8.3. Функция munmap

Прототип функции *munmap* имеет следующий вид:

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

int munmap ( caddr_t addr, int size );
```

Эта функция отсоединяет отображенную область памяти от виртуального адресного пространства процесса. Освобождаемая область начинается с виртуального адреса *addr* и заканчивается страницей памяти, имеющей виртуальный адрес *addr+size*.

В случае успешного выполнения данная функция возвращает 0, а в случае неудачи — -1.

10.8.4 Функция msync

Прототип функции *msync* выглядит следующим образом:

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

int msync ( caddr_t addr, int size, int flags );
```

Эта функция синхронизирует данные, находящиеся в отображенной области памяти, с соответствующими данными файлового объекта, имеющегося на жестком диске. Если аргумент *size* равен нулю, в отображенной области синхронизируются все измененные страницы, начинающиеся с *addr*. Если *size* больше нуля, то синхронизируются только те страницы, которые находятся в интервале от *addr* до *addr+size*.

Аргумент *flags* задает метод синхронизации и может иметь одно из следующих значений:

Значение flags	Смысл
MS_SYNC	Записать данные из отображенной области памяти на жесткий диск. Ждать завершения записи данных
MS_ASYNC	Записать данные из отображенной области памяти на жесткий диск. Не ждать завершения записи данных
MS_INVALIDATE	Аннулировать данные в отображенной области памяти. Следующее обращение к этой области приведет к выборке с жесткого диска новых страниц

В случае успешного выполнения рассматриваемая функция возвращает 0, а в случае неудачи — -1.

10.8.5. Программа типа клиент/сервер, использующая функцию mmap

Описанное в разделе 10.7.7 приложение клиент/сервер можно легко перестроить на использование функции *mmap*. Изменения необходимо внести только в заголовок *message.h*. Модули *client.C* и *server.C* не изменяются.

Новый заголовок *message4.h*, в котором используются семафоры и API *mmap*, выглядит так:

```
#ifndef MESSAGE4_H
#define MESSAGE4_H

#include <strstream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <memory.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/mman.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <sys/errno.h>
/* общие объявления для серверного процесса */
enum { MSGKEY=186, MAX_LEN=256, SHMSIZE=1024, SEMSIZE=2 };
enum { LOCAL_TIME = 1, UTC_TIME = 2, QUIT_CMD = 3, ILLEGAL_CMD = 4,
SEM_RD = 0, SEM_WR=1 };
struct mgbuf
{
   long    mtype;
   char    mtext[MAX_LEN];
};

class message
{
    private:
      int  semId;
      struct mgbuf  *msgPtr;
      enum ipc_op (   RESET_SEM, CLIENT_GET_MEM,
              CLIENT_SND_REQ, SERVER_RCV_REQ,
              SERVER_GET_MEM, SERVER_SND_RPY,
              CLIENT_RCV_RPY     };
    public:
      /* попробуем изменить значения семафоров */
      void getsem( enum ipc_op  opType )
      {
         static struct sembuf  args[2] = { (SEM_RD}, (SEM_WR} };
         switch (opType) {
```



```
            case SERVER_GET_MEM:
              return;
            case CLIENT_GET_MEM:
              args[SEM_RD].sem_op = 0,
              args[SEM_WR].sem_op = -1;
              break;
            case CLIENT_SND_REQ:
              args[SEM_RD].sem_op = 1,
              args[SEM_WR].sem_op = 0;
              break;
            case SERVER_RCV_REQ:
              args[SEM_RD].sem_op = -1,
              args[SEM_WR].sem_op = 0;
              break;
            case SERVER_SND_RPY:
              args[SEM_RD].sem_op = 1,
              args[SEM_WR].sem_op = 1;
              break;
            case CLIENT_RCV_RPY:
              args[SEM_RD].sem_op = -1,
              args[SEM_WR].sem_op = -1;
              break;
            case RESET_SEM:
              args[SEM_RD].sem_op = 0,
              args[SEM_WR].sem_op = 1;
         }
         if (semop(semId,args,SEMSIZE)==-1) perror("semop"};
      };

      /* функция-конструктор */
      message( int key )
      {
         char mfile[256], fillchr='\0';
          ostrstream(mfile,sizeof mfile} << "FOO" << key << '\0';
         int fd =open(mfile,O_RDWR,0);
         if (fd==-1) {    /* новый файл */
            if ((fd=open(mfile,O_RDWR|O_CREAT|O_TRUNC,0777}})==1)
               perror("open");
            else {
/* заполнить файл нулями; без этого в некоторых системах
   функция mmap не работает*/
               for (int i=0; i < SHMSIZE; i++} write(fd, &fillchr, 1};
               if ((semId=semget(key, SEMSIZE, IPC_CREAT|0666))==-1}
                  perror("semget");
               else getsem(RESET_SEM);  /* инициализировать новый
                                           набор семафоров */
            }
         }
         else  (  /* присоединить к существующему набору */
            if ((semId=semget(key, 0, 0))==-1) perror("semget");
            if ((msgPtr=(struct mgbuf*)mmap(0, SHMSIZE,
                PROT_READ | PROT_WRITE, MAP_SHARED, fd,0))== MAP_FAILED)
```

```
        perror("mmap");
      else close(fd);
    )
  );

  /* функция-деструктор */
  ~message() {};

  /* проверить статус создания очереди сообщений */
  int good() ( return (semId>=0) ? 1 : 0; };

  /* удалить очередь сообщений */
  int rmQ()
  {
    if (!semctl(semId,0,IPC_RMID,0) && !munmap((caddr_t)msgPtr,
                                                SHMSIZE))

       return 0;
     perror("shmctl or semctl");
    return -1;
  };

  /* передать сообщение */
  int send( const void* buf, int size, int type)
  {
    int server = (type  99);
    getsem(server ? SERVER_GET_MEM : CLIENT_GET_MEM);
    memcpy(msgPtr->mtext,buf,size);
    msgPtr->mtext[size] = '\0';
    msgPtr->mtype = type;
    getsem(server ? SERVER_SND_RPY : CLIENT_SND_REQ);
    return 0;
  };

  /* принять сообщение */
  int rcv( void* buf, int size, int type, int* rtype)
  (
      int server = (type < 0);
      getsem(server ? SERVER_RCV_REQ : CLIENT_RCV_RPY);
      memcpy(buf,msgPtr->mtext,strlen(msgPtr->mtext)+1);
      if (rtype) *rtype = msgPtr->mtype;
      if (!server) getsem(RESET_SEM);
      return strlen(msgPtr->mtext);
  );
);   /* message */
#endif
```

Изменения касаются функции-конструктора *message*, для которой вызов *shmget* заменен вызовом *mmap*. Имя отображаемого файла включает префикс имени FOO (он выбран произвольно) и заданный ключевой идентификатор. Вновь создаваемый файл инициализируется NULL-символами, количество которых определяется переменной SHMSIZE. Это необходимо для того,

чтобы убедиться: вся отображенная область памяти была выделена ядром под хранение данных.

Еще одно изменение в заголовке *message.h* коснулось функции *rmQ*. Она вызывается, когда серверный процесс завершается и ему нужно удалить свой набор семафоров и отсоединиться от отображенной области памяти.

Остальной код заголовка *message.h* — такой же, как представленный в разделе 10.7.7. Результаты выполнения этой программы также аналогичны приведенным в разделе 10.7.7.

10.9. Организация разделяемой памяти в соответствии со стандартом POSIX.1b

В POSIX.1b определены следующие API разделяемой памяти:

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

int shm_open ( char* name, int flags, mode_t mode );
int shm_unlink ( char* name );
```

Функция *shm_open* создает разделяемую область памяти, имя которой задано аргументом *name*. Значения этого аргумента такие же, как используемые для организации взаимодействия с помощью сообщений в стандарте POSIX.1b. Аргумент *flags* содержит флаги доступа к памяти (O_RDWR, O_RDONLY или O_WRONLY), а также флаги O_CREAT и O_EXCL. Аргумент *mode* используется в том случае, если данный вызов создает новую разделяемую область памяти. Его значение указывает права доступа на чтение и запись для владельца, группы и прочих пользователей, устанавливаемые для новой области.

В случае успешного выполнения эта функция возвращает неотрицательный дескриптор, а в случае неудачи — -1.

В отличие от API *shmget* ОС UNIX System V, API *shm_open* не задает размер разделяемой области памяти, который определяется в последующем вызове *ftruncate*. Прототип функции *ftruncate* выглядит так:

```
int ftruncote ( int fd, off_t shared_memory_size );
```

Здесь аргумент *fd* содержит дескриптор разделяемой памяти, возвращенный вызовом *shm_open*. Аргумент *shared_memory_size* содержит размер разделяемой области памяти, которая будет выделена. После выделения разделяемой области памяти и определения ее размера необходимо вызвать функцию *mmap*, которая отобразит разделяемую область памяти в виртуальное адресное пространство вызывающего процесса.

После завершения работы с разделяемой областью памяти процесс вызывает функцию *munmap*, которая отсоединяет разделяемую область памяти от его виртуального адресного пространства. Затем можно вызвать функцию *shm_nlink*, которая удалит разделяемую область памяти из системы. Аргументом этой функции является путевое UNIX-имя разделяемой области памяти.

Приведенная ниже программа *test_shmp.C* открывает для чтения и записи разделяемую область памяти с именем */shm.0* (посредством вызова *shm_open*) и устанавливает ее размер, вызывая для этого функцию *ftruncate*. Данная разделяемая память отображается в виртуальное адресное пространство процесса с помощью вызова *mmap*. В переменной *memp* хранится начальный адрес разделяемой области памяти.

После организации в процессе разделяемой области памяти вызывается функция *sem_init*, которая создает семафор по начальному адресу разделяемой памяти. Затем процесс работает с семафором и разделяемой областью памяти. По завершении работы процесс вызывает функцию *sem_destroy*, которая удаляет семафор из системы. После этого вызываются функции *shm_unlink* и *shm_unmap*, которые удаляют из системы разделяемую область памяти и отсоединяют ее от виртуального адресного пространства.

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <sys/mman.h>

int main()
{
   long siz = sizeof(sem_t) + 1024;
   int shmfd = shm_open ("/shm.0", O_CREAT|O_RDWR, S_IRWXU);
   if (shmfd==-1) { perror("shm_open"); return 1; }
   if (ftruncate(shmfd, siz)==-1)
      { perror("ftruncate"); return 2; }
   char* memp = (char*)mmap(0, siz, PROT_READ |PROT_WRITE,
                 MAP_SHARED, shmfd, 0L);
   if (!memp)        { perror("mmap"); return 3; }
   (void)close(shmfd);
   if (sem_init((sem_t*)memp,1,1) < 0) { perror("sem_init"); return 4;
}
   /* do work with the shared memory and semaphore */
   if (sem_destroy((sem_t*)memp) < 0) { perror("sem_destroy"); return 5; }
   if (shm_unlink("/shm.0") < 0) { perror("shm_unlink"); return 6; }
   return munmap( memp, sizeof(sem_t) + 1024);
}
```


10.9.1. Программа типа клиент/сервер, соответствующая стандарту POSIX.1b

Описанное в разделе 10.7.7 приложение клиент/сервер можно переписать, используя разделяемую память и семафоры стандарта POSIX.1b. Изменения опять-таки необходимо внести только в заголовок *message.h*. Модули *client.C* и *server.C* не изменяются.

Новый заголовок *message5.h*, который содержит класс *message*, использующий разделяемую память и семафоры стандарта POSIX.1b, выглядит так:

```
#ifndef MESSAGE5_H
#define MESSAGE5_H

/* указать, что следующий исходный текст соответствует стандарту
   POSIX.1b */
#define  _POSIX_C_SOURCE     199309L
#include <strstream.h>
#include <stdio.h>
#include <memory.h>
#include <unistd.h>
#include <string.h>
#include <limits.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <sys/mman.h>

/* общие объявления для серверного процесса */
enum { MSGKEY=186, MAX_LEN=256, MAX_MSG=20 };
enum { LOCAL_TIME=1, UTC_TIME=2, QUIT_CMD=3, ILLEGAL_CMD=4 };

/* запись данных для одного сообщения */
struct mgbuf
{
   long        mtype;              // тип сообщения
   char        mtext[MAX_LEN];     // текст сообщения
};

/* запись данных для одной разделяемой области памяти сообщения */
struct shm_header
{
   sem_t      semaphore;                  // семафор
   struct  mgbuf   msgList[MAX_MSG];      // список сообщений
};

/* класс message */
class message
{
   private:
     struct shm_header   *memptr;
     sem_t     *sem_id;
     char      mfile[256];
```

```
   enum ipc_op { GET_MEM, SND_RPY, RCV_REQ, RESET_SEM };
public:
   /* функция-конструктор */
   message( int key )
   {
      /* создать разделяемую область памяти */
      ostrstream(mfile,sizeof mfile) << "FOO" << key << '\0';
      int fd = shm_open(mfile, O_CREAT|O_RDWR,
                              S_IRWXU|S_IRWXG|S_IRWXO);
      if (fd==-1) { perror("shm_open"); return; }
      (void)ftruncate(fd,sizeof(struct shm_header));

      /* отобразить разделяемую область памяти в адресное
         пространство процесса */
      if ((memptr=(struct shm_header*)mmap(0,
            sizeof(struct shm_header),
            PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0))== MAP_FAILED) {
         perror("mmap");
         return;
      }
      close(fd);

      /* создать семафор для разделяемой области памяти */
       sem_id = (sem_t*)&memptr->semaphore;
      if ((sem_init(sem_id, 1, 1))==-1) perror("sem_init");

      /* инициализировать список сообщений как пустой */
      for (int i=0; i < MAX_MSG; i++)
         memptr->msgList[i].mtype = INT_MIN;
   };

   /* функция-деструктор: отсоединить разделяемую область
      памяти от процесса */
   ~message() { munmap(memptr, sizeof(struct shm_header)); };

   /* проверить статус создания разделяемой области памяти */
   int good() { return (memptr) ? 1 : 0; };

   /* удалить разделяемую область памяти и семафоры */
   int rmQ()
   {
      if (sem_destroy(sem_id)==-1) perror("sem_destroy");
       if (shm_unlink(mfile)==-1) perror("shm_unlink");
          return munmap(memptr, sizeof(struct shm_header));
   };

   /* попробовать изменить значение семафора */
   void getsem( enum ipc_op  opType )
   {
      switch (opType) {
         case GET_MEM:
         case RCV_REQ:
```



```
            if (sem_wait(sem_id)==-1) perror("sem_wait");
            break;
         case SND_RPY:
         case RESET_SEM:
            if (sem_post(sem_id)==-1) perror("sem_post");
            break;
      }
   };   /* getsem */

   /* передать сообщение в очередь сообщений*/
   int send( const void* buf, int size, int type)
   {
      getsem(GET_MEM);            // изменить значение семафора
      for (int i=0; i < MAX_MSG; i++)
      if (memptr->msgList[i].mtype==INT_MIN) {
    /* найти пустую позицию в очереди сообщений для сохранения
       данного сообщения */
        memcpy(memptr->msgList[i].mtext, buf, size);
        memptr->msgList[i].mtext[size] = '\0';
        memptr->msgList[i].mtype       = type;
        break;
      }
      if (i >= MAX_MSG) {    // очередь сообщений заполнена
        cerr <<  "Too many messages in the queue!\n";
      return -1;             // возвратить код неудачного завершения
      }
      getsem(SND_RPY);       // увеличить значение семафора;
      return 0;              // возвратить код успешного завершения
   };   /* send */

   /* принять сообщение */
   int rcv( void* buf, int size, int type, int* rtype)
   {
      do     {
         getsem(RCV_REQ);    // изменить значение семафора
          int lowest_type = -1;
         for (int i=0; i < MAX_MSG; i++) {
            if (memptr->msgList[i].mtype==INT_MIN) continue;
            /* закончить, если type = 0 или совпадает с типом
               сообщения */
            if (!type || type==memptr->msgList[i].mtype) break;
            /* если type < 0, найти наименьший тип сообщения < type */
            if (type < 0 && -type >= memptr->msgList[i].mtype)
               if (lowest_type==-1 || (memptr->msgList[i].mtype <
                   memptr->msgList[lowest_type].mtype))
                  lowest_type = i;
         }
            if (i < MAX_MSG || lowest_type !=-1) {  // найти одно
                                                    // сообщение
            if (lowest_type!=-1) i = lowest_type;
            /* скопировать текст сообщения и его тип в переменные
               вызывающего процесса */
```

```
            memcpy(buf,memptr->msgList[i].mtext,
                   strlen(memptr->msgList[i].mtext)+1);
            if (rtype) *rtype = memptr->msgList[i].mtype;
            /* отметить позицию очереди как пустую */
            memptr->msgList[i].mtype = INT_MIN;
            getsem(RESET_SEM); // увеличить значение семафора
            return strlen((char*)buf); // возвратить размер
                                       // сообщения
        }
        getsem(RESET_SEM);          // увеличить значение семафора
        sleep(1);                   // подождать 1 секунду
        } while(1);                 // проверить очередь повторно
    };  /* rcv */
}; /* message */
#endif
```

Новый класс *message* отличается от класса *message*, представленного в разделе 10.7.7. Это обусловлено тем, что семафоры POSIX.1b принимают целые значения, которые могут всякий раз изменяться только на 1. Значения же семафоров System V разрешается изменять на любую целую величину. Вследствие этого ограничения новый класс *message* не поддерживает взаимодействие между сервером и одним из клиентов в случае, если остальные клиентские процессы блокируются своими собственными вызовами *semop*. В результате получается новый класс *message*, код которого проще: каждый запрос *send* и *receive* предваряется вызовом *sem_wait* для получения совместно используемого семафора. Этот вызов завершается вызовом *sem_post*, который освобождает данный семафор для разблокирования других процессов (серверных и клиентских), требующих доступа к очереди сообщений. Кроме того, класс *message* теперь полнее реализует возможности, заложенные в механизмах обмена сообщениями, используемых в System V и POSIX.1b.

Функция-конструктор этого класса получает в качестве аргумента целочисленный ключ и задает имя для разделяемой области памяти. Эта область памяти выделяется посредством вызова *shm_open*. Процесс может читать данные из вновь выделенной области и записывать их в нее, а для владельца, группы и прочих пользователей устанавливаются права доступа на чтение, запись и выполнение (в случае, если до вызова данная область памяти не существует).

Размер выделенной разделяемой области памяти с помощью вызова *ftruncate* устанавливается равным размеру, заданному в *struct shm_header*. Эта структура определяет все поля данных для одной разделяемой области памяти — значения совместно используемого семафора и список записей, в которых хранятся сообщения сервера и клиента.

Разделяемая область памяти отображается в виртуальное адресное пространство процесса при помощи функции *mmap*. Начальный адрес отображаемой памяти определяется ядром. После вызова *mmap* дескриптор файла, возвращенный вызовом *shm_open*, закрывается (он больше не нужен). Затем с помощью вызова *sem_open* создается семафор. Этот новый семафор

размещается в начале совместно используемой области памяти, и его начальное значение устанавливается равным 1.

Наконец, список сообщений инициализируется установкой типа каждого сообщения в INT_MIN (большое отрицательное число). Такая установка показывает, что они не используются.

Когда сообщение передается в очередь функцией *message::sent*, сначала посредством вызова функции *message::getsem* изменяется значение семафора (эта функция вызывает *sem_wait*). В результате этого вызова процесс открывает семафор и теперь может обращаться к списку сообщений, находящихся в очереди. Он просматривает каждый элемент списка сообщений, пока не находит первый свободный элемент списка (тип которого INT_MIN). Процесс заносит в эту запись данные сообщения (текст и тип). После этого процесс освобождает семафор еще одним вызовом функции *message::getsem* (которая вызывает *sem_post*).

Когда процесс пытается принять сообщение из очереди с помощью функции *message::rcv*, сначала посредством функции *message::getsem* изменяется значение семафора (названная функция вызывает *sem_wait*). В результате такого вызова процесс открывает семафор и теперь может обращаться к списку сообщений, находящихся в очереди. Он просматривает каждый элемент списка сообщений, пока не находит запись, тип которой совпадает с типом сообщения, заданным вызывающим процессом. Процесс копирует текст и тип сообщения, содержащиеся в этой записи, в переменные, являющиеся аргументами функции, и закрывает семафор (посредством вызова функции *message::getsem*). Наконец, функция *message::rcv* возвращает вызывающему процессу размер сообщения. Если ни у одного сообщения, поступившего в очередь, тип не совпадает с типом, заданным вызывающим процессом, функция закрывает семафор, переводит процесс на одну секунду в состояние ожидания, а затем повторяет весь процесс получения сообщения. Таким образом вызывающий процесс блокируется до тех пор, пока в очередь не поступит сообщение, удовлетворяющее критериям поиска.

Функция *message::rmQ* вызывается для удаления разделяемой области памяти и семафора. Эта задача реализуется путем вызова *sem_destroy* (для удаления семафора), *shm_unlink* (для удаления разделяемой области памяти) и, наконец, *munmap* (для отсоединения разделяемой области памяти от виртуального адресного пространства процесса).

Новый заголовок *message.h* можно включить при компиляции клиентской и серверной программ так, как показано в разделе 10.3.7. Результат выполнения новых программ должен совпадать с результатом, приведенным в разделе 10.3.7.

10.10. Заключение

В этой главе рассматриваются методы межпроцессного взаимодействия, используемые в UNIX System V.3, V.4 и POSIX.1b: сообщения, семафоры, разделяемая память и API *mmap*. Подробно освещается синтаксис этих API и предлагаются примеры программ, иллюстрирующие их использование. Общий недостаток перечисленных методов IPC состоит в том, что не существует стандартов, которые определяли бы порядок их использования для межмашинного взаимодействия. В следующей главе будут рассмотрены такие методы межпроцессного взаимодействия, как гнезда (BSD UNIX) и TLI, интерфейс транспортного уровня (UNIX System V.3 и V.4). Гнезда и TLI позволяют обеспечить взаимодействие между процессами, которые выполняются как на одной машине, так и на разных.

ГЛАВА 11

Гнезда и интерфейс транспортного уровня

В предыдущей главе были рассмотрены методы межпроцессного взаимодействия, которые основаны на использовании сообщений, разделяемой памяти и семафоров. Эти методы хорошо приспособлены для организации взаимодействия процессов на одном компьютере, но не могут обеспечить взаимодействие процессов, работающих на разных машинах. Связано это с тем, что очереди сообщений, области разделяемой памяти и наборы семафоров обозначаются целочисленными ключами. Эти ключи уникальны только для отдельных компьютеров, но не для множества машин. Поэтому процесс, работающий на компьютере A, не может обратиться к очереди сообщений, созданной на машине B, с помощью только ключа этой очереди. Методы IPC, определенные стандартом POSIX.1b, позволяют решить данную проблему путем использования для обозначения сообщений, семафоров и разделяемой памяти текстовых имен. Этот стандарт предоставляет производителям систем право самим определять и интерпретировать указанные имена таким образом, чтобы механизм IPC мог работать и в компьютерной сети.

В системе BSD UNIX 4.2 были впервые введены гнезда (sockets), которые предоставляют независимые от протокола услуги по организации сетевого интерфейса и способны обеспечить работоспособность методов IPC в масштабах локальной сети. В частности, гнезда могут работать как с протоколом TCP (Transmission Control Protocol), так и с протоколом UDP (User Datagram Protocol). Обращаться к гнезду можно по IP-адресу хост-машины и номеру порта. Заданный таким образом адрес уникален в масштабах всей Internet, так как для каждой машины комбинация адреса и номера порта уникальна. Следовательно, два процесса, выполняемых на отдельных машинах, могут взаимодействовать друг с другом через гнезда.

Гнезда нашли широкое применение во многих сетевых приложениях. Сейчас они используются в BSD UNIX 4.3, 4.4 и даже в UNIX System V.4. При этом, однако, в UNIX System V.4 гнезда реализованы несколько по-иному, нежели в BSD UNIX. Эти различия описываются в следующих разделах.

Интерфейс транспортного уровня (Transport Level Interface, TLI) был разработан в UNIX System V.3. Он стал ответом System V на появление гнезд в BSD UNIX. Методика его использования и соответствующие API похожи на методику использования и API гнезд. Более того, поскольку TLI был разработан на базе механизма STREAMS, он поддерживает большинство транспортных протоколов и более гибок, чем гнезда. Этот интерфейс используется и в UNIX System V.3, и в UNIX System V.4. В стандарте X/Open TLI называется XTI (X/Open Transport Interface — транспортный интерфейс X/Open).

В следующих разделах данной главы рассматриваются и API гнезд, и TLI. Приводятся примеры приложений, реализованных на их базе. Следует отметить, что в стандарте POSIX гнезда и TLI не определены.

11.1. Гнезда

Различают гнезда *с установлением соединения* (т.е. адреса гнезд отправителя и получателя выясняются заранее, до передачи сообщений между ними) и *без установления соединения* (адреса гнезд отправителя и получателя передаются с каждым сообщением, посылаемым из одного процесса в другой). В зависимости от того, к какому домену принадлежит гнездо, используются разные форматы адресов гнезд и базовые транспортные протоколы. При использовании гнезд обычно применяются следующие стандартные домены: AF_UNIX (формат адреса — путевое имя UNIX) и AF_INET (формат адреса — хост-имя и номер порта).

Для каждого гнезда назначается тип, посредством которого определяется способ передачи данных между двумя гнездами. Если тип гнезда — *виртуальный канал* (virtual circuit), то данные передаются последовательно, с достаточной степенью надежности и не дублируются. Если тип гнезда — *дейтаграмма* (datagram), то условие последовательности пересылки данных не выполняется и надежность их передачи низкая. Тип гнезда с установлением соединения — как правило, виртуальный канал, а тип гнезда без установления соединения — дейтаграмма. Дейтаграммные гнезда обычно работают быстрее, чем виртуальные каналы (потоковые гнезда), и используются в приложениях, где быстродействие важнее, чем надежность.

Гнездо каждого типа поддерживает один или несколько транспортных протоколов, однако в любой UNIX-системе для любого гнезда всегда указывается протокол по умолчанию. Протокол по умолчанию для виртуального канала — TCP, а для дейтаграммы — UDP.

Гнезда, которые используются для связи компьютеров друг с другом, должны быть одного типа и относиться к одному домену. Кроме того, гнезда с установлением соединения взаимодействуют по схеме клиент/сервер:

серверному гнезду назначается общеизвестный адрес, и оно непрерывно ожидает прибытия клиентских сообщений. Клиентский процесс посылает сообщения на сервер по объявленному адресу серверного гнезда. Назначать адреса клиентским гнездам не нужно, потому что обычно ни один серверный процесс сообщения клиентам таким способом не передает.

Гнезда без установления соединения, с другой стороны, взаимодействуют по одноранговой схеме: каждому гнезду назначается адрес, и процесс может посылать сообщения другим процессам, используя адреса их гнезд.

Интерфейсы прикладного программирования гнезд перечислены ниже :

API гнезд	Назначение
socket	Создает гнездо заданного типа и с указанным протоколом для конкретного домена
bind	Присваивает гнезду имя
listen	Задает количество ожидающих клиентских сообщений, которые можно поставить в очередь к одному серверному гнезду
accept	Принимает запрос на соединение от клиентского гнезда
connect	Посылает запрос на соединение в серверное гнездо
send, sendto	Передает сообщение в удаленное гнездо
recv, recvfrom	Принимает сообщение из удаленного гнезда
shutdown	Закрывает гнездо для чтения и/или записи

Последовательность вызовов API гнезд, которые устанавливают между клиентом и сервером соединение типа виртуальный канал, представлена на рис. 11.1.

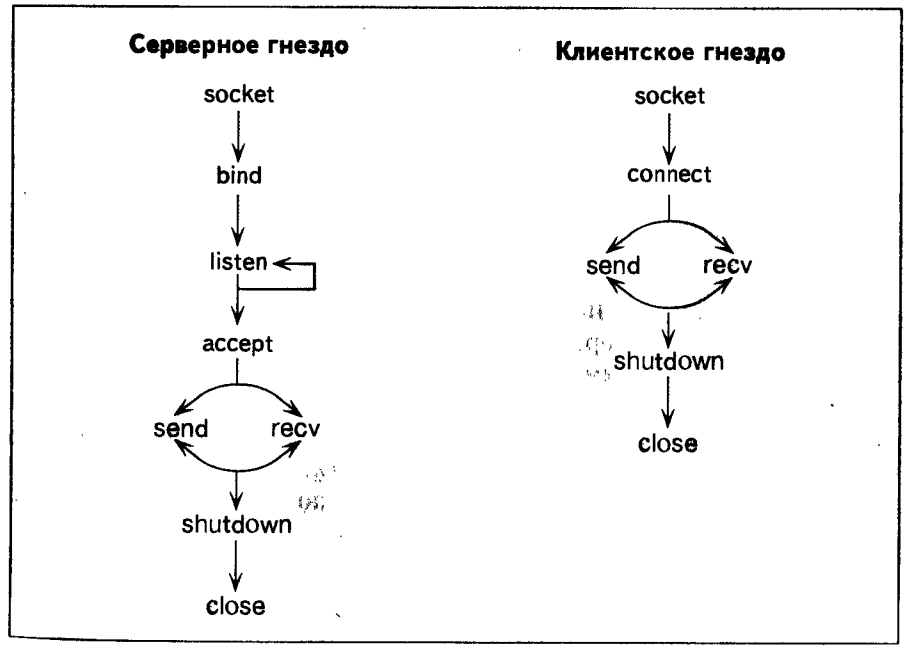


Рис. 11.1. Последовательность вызовов API для серверного и клиентского гнезд

Чтобы понять смысл использования этих API, представьте, что гнездо — это телефонный аппарат. API *socket* предназначен для покупки телефона в магазине. API *bind* присваивает телефону номер. API *listen* просит вашу телефонную компанию подключить телефон к сети. API *connect* звонит кому-то с вашего телефона. API *answer* отвечает на телефонный звонок. API *send* разговаривает по телефону. Наконец, API *shutdown* кладет трубку после завершения разговора. Чтобы отказаться от услуг телефонной компании, используйте API *close* с дескриптором гнезда, возвращенным из вызова функции *socket*.

На стороне клиента процесс вызывает функцию *socket* для установки телефона. Затем он вызывает функцию *connect* и с ее помощью набирает номер сервера, после чего посредством вызова функций *send* и *recv* общается с сервером. По окончании разговора процесс вызывает функцию *shutdown*, которая дает сигнал отбоя, и функцию *close*, уничтожающую "телефон".

Последовательность вызовов API, создающих для обеспечения межпроцессного взаимодействия дейтаграммные гнезда, изображена на рис. 11.2.

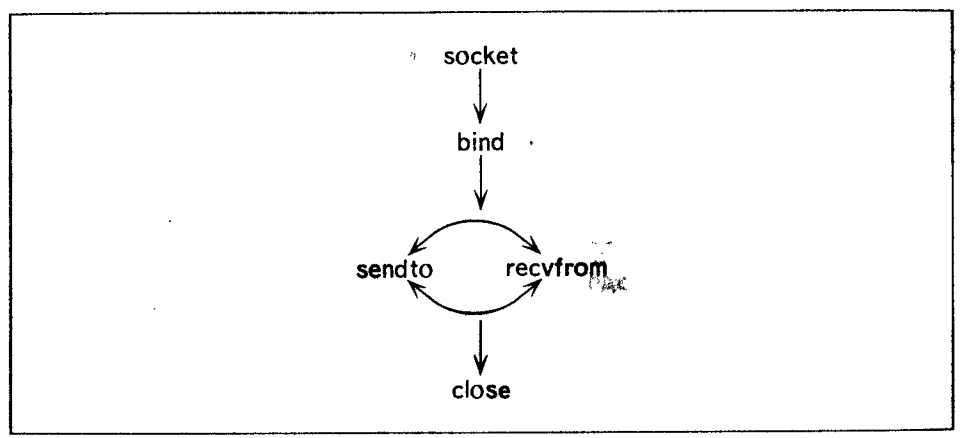


Рис. 11.2. Последовательность вызовов API, обеспечивающих взаимодействие процессов через дейтаграммные гнезда

Работать с дейтаграммными гнездами достаточно легко. Процесс вызывает функцию *socket*, которая создает гнездо, а затем с помощью функции *bind* присваивает гнезду имя. Затем процесс вызывает функцию *sendto* для передачи сообщений в другие процессы. Каждое сообщение снабжается адресом гнезда получателя. Процесс получает также сообщения из других процессов посредством вызова функции *recvfrom*. Каждое полученное сообщение содержит адрес гнезда отправителя, что позволяет процессу безошибочно отправить ответ.

По завершении межпроцессного взаимодействия процесс вызывает функцию *close*, которая удаляет гнездо. Вызывать функцию *shutdown* не нужно, потому что виртуальный канал, обеспечивающий взаимодействие с другими процессами, не организовывался.

В последующих разделах более подробно рассматривается синтаксис API гнезд и методика их использования.

11.1.1. Функция socket

Прототип функции *socket* выглядит следующим образом:

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

int socket ( int domain, int type, int protocol );
```

Эта функция создает для указанного пользователем домена гнездо заданного типа и с указанным протоколом.

Аргумент *domain* определяет правила именования гнезда и формат адреса, используемые в протоколе. Широко применяются такие домены, как AF_UNIX (домен UNIX) и AF_INET (Internet-домен).

Аргумент *type* задает тип гнезда. Возможные значения этого аргумента приведены ниже.

Тип гнезда	Смысл
SOCK_STREAM	Сообщения передаются в виде упорядоченного двунаправленного потока байтов с высокой степенью надежности и предварительным установлением соединения
SOCK_DGRAM	Межпроцессное взаимодействие обеспечивается с помощью дейтаграмм. Сообщения передаются быстро (как правило, без установления соединения), но с низкой степенью надежности
SOCK_SEQPACKET	Двунаправленная последовательная высоконадежная передача сообщений фиксированной максимальной длины с предварительным установлением соединения

Аргумент *protocol* указывает конкретный протокол, который следует использовать с данным гнездом. Фактическое значение этого аргумента зависит от значения аргумента *domain*. Как правило, оно устанавливается в 0, и ядро само выбирает для указанного домена соответствующий протокол.

В случае успешного выполнения рассматриваемая функция возвращает целочисленный дескриптор гнезда, а в случае неудачи возвращает -1. Отметим, что дескриптор гнезда — это то же самое, что и дескриптор файла; он занимает одну ячейку таблицы дескрипторов файлов в вызывающем процессе.

11.1.2. Функция bind

Прототип функции *bind* выглядит следующим образом:

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

int bind ( int sid, struct sockaddr* addr_p, int len );
```

Эта функция присваивает гнезду имя. Гнездо обозначается аргументом *sid*, значение которого, возвращенное функцией *socket*, представляет собой дескриптор гнезда. Аргумент *addr_p* указывает на структуру, содержащую имя, которое должно быть присвоено гнезду. Аргумент *len* задает размер структуры, на которую указывает аргумент *addr_p*.

В каждом домене используется своя структура объекта, на который указывает аргумент *addr_p*. В частности, в случае гнезда домена UNIX присваиваемое имя представляет собой путевое UNIX-имя, а структура объекта, на который указывает аргумент *addr_p*, имеет такой вид:

```
struct sockaddr
{
    short     sun_family;
    char     sun_path[];
};
```

Здесь полю *sun_family* следует присвоить значение AF_UNIX, а поле *sun_path* должно содержать путевое UNIX-имя. При успешном выполнении вызова *bind* в файловой системе создается файл с именем, заданным в поле *sun_path*. Если гнездо больше не нужно, этот файл следует удалить с помощью API *unlink*.

В случае гнезда домена Internet присваиваемое имя состоит из хост-имени машины и номера порта, а структура объекта, на который указывает аргумент *addr_p*, имеет такой вид:

```
struct sockaddr_in
{
    short           sin_family;
    u_short         sin_port;
    struct in_addr  sin_addr;
};
```

Здесь полю *sin_family* следует присвоить значение AF_INET. Поле *sin_port* — это номер порта, а поле *sin_addr* — имя хост-машины, для которой создается гнездо. Структура *sockaddr_in* определяется в заголовке <netinet/in.h>.

При успешном выполнении эта функция возвращает 0, а в случае неудачи возвращает -1.

11.1.3. Функция listen

Прототип функции *listen* выглядит следующим образом:

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

int listen ( int sid, int size );
```

Эта функция вызывается серверным процессом для создания гнезда, ориентированного на установление соединения (типа SOCK_STREAM или SOCK_SEQPACKET).

Аргумент *sid* представляет собой дескриптор гнезда, возвращенный функцией *socket*. Аргумент *size* задает максимальное число запросов на установление соединения, которые могут быть поставлены в очередь к данному гнезду. В большинстве UNIX-систем максимально допустимое значение аргумента *size* — 5.

При успешном выполнении эта функция возвращает 0, а в случае неудачи возвращает -1.

11.1.4. Функция connect

Прототип функции *connect* выглядит так:

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

int connect ( int sid, struct sockaddr* addr_p, int len );
```

Эта функция вызывается в клиентском процессе для установления соединения с серверным гнездом.

Аргумент *sid* представляет собой дескриптор гнезда, возвращенный функцией *socket*. В BSD 4.2 и 4.3 имя гнезда, указанное аргументом *sid*, совпадает с именем используемого транспортного протокола. В System V.4 имя гнезду присваивается транспортным протоколом.

Аргумент *addr_p* — это указатель на адрес объекта типа *struct sockaddr*, хранящего имя серверного гнезда, с которым должно быть установлено соединение. Фактически структура этого объекта зависит от домена, на основе которого создается гнездо. Возможный формат — *struct sockaddr* (для домена UNIX) или *struct sockaddr_in* (для домена Internet).

Аргумент *len* задает размер объекта (в байтах), на который указывает аргумент *addr_p*.

Если *sid* обозначает потоковое гнездо, то между клиентским и серверным гнездами устанавливается соединение с использованием виртуального канала.

Потоковое гнездо клиента может соединяться с гнездом сервера только один раз. Если *sid* обозначает дейтаграммное гнездо, то для всех последующих вызовов функции *send*, осуществляемых через это гнездо, устанавливается адрес по умолчанию. Дейтаграммное гнездо может соединяться с гнездом сервера многократно, изменяя установленные по умолчанию адреса. Путем соединения с гнездом, имеющим NULL-адрес, дейтаграммные гнезда могут разорвать соединение.

При успешном выполнении эта функция возвращает 0, а в случае неудачи — -1.

11.1.5. Функция accept

Прототип функции *accept* выглядит следующим образом:

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

int accept ( int sid, struct sockaddr* addr_p, int len_p );
```

Эта функция вызывается в серверном процессе для установления соединения с клиентским гнездом (которое делает запрос на установление соединения посредством вызова функции *connect*).

Аргумент *sid* представляет собой дескриптор гнезда, возвращенный функцией *socket*. Аргумент *addr_p* — это указатель на адрес объекта типа *struct sock*; в нем хранится имя клиентского гнезда, с которым устанавливает соединение серверное гнездо.

Аргумент *len_p* изначально устанавливается равным максимальному размеру объекта, указанному аргументом *addr_p*. При возврате он содержит размер имени клиентского гнезда, на которое указывает аргумент *addr_p*.

Если аргумент *addr_p* или аргумент *len_p* имеет значение NULL, эта функция не передает имя клиентского гнезда обратно в вызывающий процесс.

В случае неудачи рассматриваемая функция возвращает -1. В противном случае она возвращает дескриптор нового гнезда, с помощью которого серверный процесс может взаимодействовать исключительно с данным клиентом.

11.1.6. Функция send

Прототип функции *send* выглядит следующим образом:

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

int send ( int sid, const char* buf, int len, int flag );
```


Эта функция передает содержащееся в аргументе *buf* сообщение длиной *len* байтов в гнездо, заданное аргументом *sid* и соединенное с данным гнездом.

Аргументу *flag* обычно присваивается значение 0, но он может иметь и значение MSG_OOB. В этом случае сообщение, содержащееся в *buf*, должно быть передано как высокоприоритетное (out-of-band message).

Через гнезда можно передавать сообщения двух типов: обычные и высокоприоритетные. По умолчанию все сообщения, передаваемые гнездом, являются обычными, если явно не указано, что они высокоприоритетные. Если из гнезда передается более одного сообщения одного типа, другое гнездо принимает их по алгоритму FIFO. Гнездо-получатель может выбрать тип сообщений, которые оно хотело бы получать. Сообщения с высоким приоритетом следует использовать только в экстренных случаях.

Если процесс пользуется гнездом с установлением соединения или гнездом без установления соединения, для которого указан адрес получателя по умолчанию (посредством вызова функции *connect*), он может передавать обычные сообщения с помощью либо API *send*, либо API *write*. При этом функции *send* и *sendto* можно использовать для передачи сообщений нулевой длины, а *write* — нельзя. Кроме того, в BSD 4.2 и 4.3 функция *write* при обращении к гнезду, соединение с которым не установлено, дает сбой. В System V.4 такой вызов вроде бы выполняется успешно, но никакие данные не передаются.

В случае неудачи эта функция возвращает -1; в случае успешного выполнения возвращается число переданных байтов данных.

11.1.7. Функция sendto

Прототип функции *sendto* выглядит следующим образом:

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

int sendto ( int sid, const char* buf, int len, int flag,
                    struct sockaddr* addr_p, int* len_p );
```

Эта функция делает то же самое, что и API *send*, только вызывающий процесс указывает также адрес гнезда-получателя (в аргументах *addr_p* и *len_p*).

Аргументы *sid*, *buf*, *len* и *flag* — те же самые, что в API *send*. Аргумент *addr_p* — это указатель на объект, который содержит имя гнезда-получателя. Аргумент *len_p* содержит число байтов в объекте, на который указывает аргумент *addr_p*.

В случае неудачи данная функция возвращает -1; в случае успешного выполнения возвращается число переданных байтов данных.

11.1.8. Функция recv

Прототип функции *recv* выглядит следующим образом:

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

int recv ( int sid, char* buf, int len, int flag );
```

Эта функция принимает сообщение через гнездо, указанное в аргументе *sid*. Принятое сообщение копируется в буфер *buf*, а максимальный размер *buf* задается аргументом *len*.

Если в аргументе *flag* указан флаг MSG_OOB, то приему подлежит высокоприоритетное сообщение. В противном случае ожидается обычное сообщение. Кроме того, в аргументе *flag* может быть указан флаг MSG_PEEK, означающий, что процесс желает "взглянуть" на полученное сообщение, но не собирается удалять его из потокового гнезда. Такой процесс может повторно вызвать функцию *recv* для приема сообщения позже.

Если процесс пользуется гнездом (с установлением соединения или без установления соединения), для которого указан адрес получателя по умолчанию (посредством вызова API *bind*), он может принимать обычные сообщения через это гнездо с помощью либо API *recv*, либо API *read*. В BSD 4.2 и 4.3 функция *read* при использовании с гнездом, с которым не установлено соединение, дает сбой. В System V.4 в подобном случае функция *read* возвращает нулевое значение в блокирующем режиме и -1 в неблокирующем.

В случае неудачи функция *recv* возвращает -1; в случае успешного выполнения возвращается число байтов данных, записанных в буфер *buf*.

11.1.9. Функция recvfrom

Прототип функции *recvfrom* выглядит следующим образом:

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

int recvfrom ( int sid, const char* buf, int len, int flag,
               struct sockaddr* addr_p, int* len_p );
```

Эта функция делает то же самое, что и API *recv*, только при ее вызове задаются аргументы *addr_p* и *len_p*, позволяющие узнать имя гнезда-отправителя.

Аргументы *sid*, *buf*, *len* и *flag* — те же самые, что в API *recv*. Аргумент *addr_p* — это указатель на объект, который содержит имя гнезда-отправителя.

Аргумент *len_p* сообщает число байтов в объекте, на который указывает аргумент *addr_p*.

В случае неудачи функция *recvfrom* возвращает -1; в случае успешного выполнения возвращается число принятых байтов данных.

11.1.10. Функция shutdown

Прототип функции *shutdown* выглядит следующим образом:

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

int shutdown ( int sid, int mode );
```

Данная функция закрывает соединение между серверным и клиентским гнездами.

Аргумент *sid* — это дескриптор гнезда, возвращенный функцией *socket*. Аргумент *mode* задает режим закрытия. Вот его возможные значения:

Режим	Смысл
0	Закрывает гнездо для чтения. При попытке продолжить чтение будут возвращаться нулевые байты (EOF)
1	Закрывает гнездо для записи. Дальнейшие попытки передать данные в это гнездо приведут к выдаче кода неудачного завершения, -1
2	Закрывает гнездо для чтения и записи. Дальнейшие попытки передать данные в это гнездо приведут к выдаче кода неудачного завершения -1, а при продолжении чтения будет возвращаться нулевое значение (EOF)

В случае неудачи данная функция возвращает -1, а в случае успешного выполнения — нуль.

11.2. Создание потоковых гнезд

В этом разделе описана пара программ типа клиент/сервер, на примере которых показывается, как нужно создавать потоковые гнезда для IPC. Используемые здесь потоковые гнезда могут быть гнездами домена UNIX или гнездами домена Internet. В последнем случае клиентский и серверный процессы могут выполняться как на одной машине, так и на разных.

Чтобы обеспечить реализацию приложений, функционирующих на основе гнезд, определяется класс *sock*, который инкапсулирует API гнезд из прикладных программ. Преимущества этого подхода таковы:

1. В прикладных программах можно обходиться без адресов гнезд. Пользователи таких прикладных программ манипулируют адресами гнезд путем указания имен гнезд или хост-имен и номеров портов.

2. Функции-члены *read* и *write* класса *sock* аналогичны одноименным файловым API ОС UNIX.

Указанные особенности позволяют сократить расходы на изучение методики применения гнезд, а также сэкономить время, затрачиваемое на программирование.

Класс *sock* определяется в заголовке *sock.h*:

```
#ifndef SOCK_H
#define SOCK_H

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <memory.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/systeminfo.h>
const int BACKLOG_NUM = 5;

class sock
{
   private:
     int sid;          // дескриптор гнезда
     int domain;       // домен гнезда
     int socktype;     // тип гнезда
     int rc;           // код возврата функции-члена

     /* создать имя Internet-гнезда на основании хост-имени
        и номера порта */
      int constr_name( struct sockaddr_in& addr, const char* hostnm,
                       int port )
      {
        addr.sin_family = domain;
        if (!hostnm)
           addr.sin_addr.s_addr = INADDR_ANY;
        else {
           struct hostent *hp = gethostbyname(hostnm);
           //if (hp==0) {
              // perror("gethostbyname"); return -1;
           //}
           memcpy((char*)&addr.sin_addr,(char*)hp->h_addr,
                          hp->h_length);
        }
        addr.sin_port = htons(port);
        return sizeof(addr);
      };
```



```
     /* сформировать имя гнезда домена UNIX на основании путевого
        имени */
     int constr_name( struct sockaddr& addr, const char* Pathnm )
     {
        addr.sa_family = domain;
        strcpy(addr.sa_data, Pathnm );
        return sizeof(addr.sa_family) + strlen(Pathnm) + 1;
     };

     /* преобразовать IP-адрес в хост-имя */
     char* ip2name( const struct in_addr in )
     {
        u_long laddr;
        if ((int)(laddr = inet_addr(inet_ntoa(in))) == -1) return 0;
        struct hostent *hp = gethostbyaddr((char *)&laddr,
        sizeof (laddr), AF_INET);
        if (hp == NULL) return 0;
        for (char **p = hp->h_addr_list; *p != 0; p++) {
           (void) memcpy((char*)&in.s_addr, *p, sizeof (in.s_addr));
           if (hp->h_name) return hp->h_name;
         }
         return 0;
     };
   public:
     sock( int dom, int type, int protocol=0 ) : domain(dom),
socktype(type)
     {
         if ((sid=socket(domain=dom, socktype=type,protocol))<0)
           perror("socket");
     };
     ~sock()  ( shutdown(); close(sid);  };// деструктор
     int fd()  ( return sid; };  /* возвращает дескриптор гнезда */
     int good() { return sid >= 0;  };  /* статус объекта sock */
     int bind( const char* name, int port=-1 ) ( /* присвоение
                                                    UNIX-имени */
        if (port == -1) (    // гнездо домена UNIX
           struct sockaddr addr;
           int len = constr_name( addr, name);
           if ((rc= ::bind(sid,&addr,len))<0) perror("bind");
         } else {
           struct sockaddr_in addr;
           int len = constr_name( addr, name, port);
           if ((rc= ::bind(sid, (struct sockaddr *)&addr, len))<0  ||
              (rc=getsockname(sid, (struct sockaddr*)&addr, &len))<0)
             perror("bind or getsockname");
           else cerr << "Socket port: " << ntohs(addr.sin_port) << endl;
        }
        if (rc!=-1 && socktype!=SOCK_DGRAM && (rc=listen
                                                 (sid,BACKLOG_NUM)) < 0)
           perror("listen");
      return rc;
        };
```

```
int accept ( char* name, int* port_p)
{
   if (!name) return ::accept(sid, 0, 0);
   if (!port_p || *port_p == -1) ( // гнездо домена UNIX
       struct sockaddr addr;
       int size = sizeof(addr);
       if ((rc = ::accept(sid, &addr, &size)) >-1)
     strncpy(name,addr.sa_data,size), name[size]='\0';
     } else (
  struct sockaddr_in addr;
  int size = sizeof (addr);
       if ((rc = ::accept( sid, (struct sockaddr*)&addr,
            &size)) >-1)
       (
if (name) strcpy(name,ip2name(addr.sin_addr));
if (port_p) *port_p = ntohs(addr.sin_port);
       }
    }
    return rc;
};
int connect( const char* hostnm, int port=-1 )
                          /* соединение с гнездом UNIX */
(
   if (port==-1) (          /* гнездо домена UNIX */
      struct sockaddr addr;
      int len = constr_name( addr, hostnm);
      if ((rc= ::connect(sid,&addr,len))<0) perror("bind");
   ) else {
      struct sockaddr_in addr;
      int len = constr_name( addr, hostnm, port);
      if ((rc= ::connect(sid,(struct sockaddr *)&addr,len))
           <0) perror("bind");
   }
   return rc;
};
int write( const char* buf, int len, int flag=0,
           int nsid=-1 )          /* передать сообщение */
{
    return ::send(nsid==-1 ? sid : nsid, buf, len, flag );
};
int read( char* buf, int len, int flag=0, int nsid=-1 )
                                  /* прочитать сообщение */
{


    return ::recv(nsid==-1 ? sid : nsid, buf, len, flag );
};
/* запись в гнездо домена UNIX с указанным именем */
int writeto( const char* buf, int len, int flag,
             const char* name, const int port, int nsid=-1 )
(
    if (port==-1) (
```



```
          struct sockaddr addr;
              int size = constr_name( addr, name);
              return ::sendto(nsid==-1 ? sid : nsid, buf, len,
                              flag, &addr, size );
          } else (
           struct sockaddr_in addr;
              char buf1[80];
              if (!name) (    /* использовать имя локального
                                 хоста, если не указано другое */
                 if (sysinfo(SI_HOSTNAME,buf1,sizeof buf1)==-1L)
                     perror("sysinfo");
                 name = buf1;
              }
              int size = constr_name( addr, name, port);
              return ::sendto(nsid==-1 ? sid : nsid, buf, len, flag,
                   (struct sockaddr*)&addr, size );
          }
  };
  /* получить сообщение из гнезда домена UNIX */
       int readfrom( char* buf, int len, int flag, char* name,
                     int *port_p, int nsid =-1)
       (
          if (!port_p || *port_p == -1) ( // гнездо домена UNIX
             struct sockaddr addr;
             int size = sizeof(addr);
             if ((rc=::recvfrom(nsid==-1 ? sid : nsid, buf, len,
                  flag, &addr, &size)) >-1
                     && name)
                strncpy(name,addr.sa_data,rc), name[rc]='\0';
          ) else (
       struct sockaddr_in addr;
       int size = sizeof (addr);
            if ((rc = ::recvfrom(nsid==-1 ? sid : nsid, buf, len, flag,
                           (struct sockaddr*)&addr, &size)) >-1)
            (
    if (name) strcpy(name,ip2name(addr.sin_addr));
    if (port_p) *port_p = ntohs(addr.sin_port);
            }
         )
         return rc;
  };
       int shutdown( int mode = 2 )    // закрыть гнездо
       (
            return ::shutdown (sid,mode);
       };
);      /* класс sock */

#endif
```

Класс *sock* позволяет скрыть от прикладных программ низкоуровневый API гнезд. Приложение, которому необходимо открыть гнездо домена UNIX, должно указать функции-члену *bind* или *connect* лишь путевое UNIX-имя.

С другой стороны, если приложению необходимо открыть гнездо домена Internet, оно должно указать только хост-имя и номер порта. Манипулировать какими-либо объектами типа *struct sockaddr* приложению не придется. Это экономит время программиста и уменьшает число ошибок, могущих возникнуть при создании адресов гнезд.

Функции-члены класса *sock* почти в точности соответствуют API гнезд. Это облегчает задачу тем пользователям, которым нравится переключаться с API гнезд на объекты класса *sock* и наоборот. В функциях *sock::read*, *sock::write*, *sock::readfrom* и *sock::writeto* используется аргумент *nsid*, значение которому присваивается в том случае, если вызывающий процесс является сервером. Такой процесс может взаимодействовать с клиентским процессом через дескриптор гнезда *nsid*, полученный в результате вызова функции *sock::accept*.

Ниже показана серверная программа, в которой класс *sock* используется для создания потокового гнезда, устанавливающего соединение с гнездом клиентской программы:

```
#include "sock.h"

const char* MSG2 = "Hello MSG2";
const char* MSG4 = "Hello MSG4";

main( int argc, char* argv[])
{
   char  buf[80], socknm[80];
   int  port=-1, nsid, rc;

   if (argc < 2) {
      cerr << "usage: " << argv[0] << " <sockname|port> [<host>]\n";
      return 1;
   }

   /* проверить, указан ли номер порта гнезда */
   (void)sscanf(argv(1],"%d",&port);

   /* создать потоковое гнездо */
   sock sp( port!=-1 ? AF_INET : AF_UNIX, SOCK_STREAM );
   if (!sp.good()) return 1;

   /* присвоить имя серверному гнезду */
   if (sp.bind(port==-1 ? argv[1] : argv(2],port) < 0) return 2;

   /* принять запрос на соединение от клиентского гнезда */
   if ((nsid = sp.accept(0, 0)) < 0) return 1;

   /* прочитать сообщение MSG1 из клиентского гнезда */
   if ((rc=sp.read(buf, sizeof buf, 0, nsid)) < 0) return 5;
   cerr << "server: receive msg: '" << buf << "'\n";

   /* записать сообщение MSG2 в клиентское гнездо */
   if (sp.write(MSG2,strlen(MSG2)+1,0,nsid)<0) return 6;
```



```
   /* прочитать сообщение MSG3 из клиентского гнезда */
   if (sp.readfrom( buf, sizeof buf, 0, socknm, &port, nsid) > 0)
   cerr << "server: recvfrom '" << socknm << "' msg: " << buf << endl;

   /* записать сообщение MSG4 в клиентское гнездо */
   if (write(nsid,MSG4,strlen(MSG4)+1)==-1) return 7;
}
```

Аргументом командной строки для этой серверной программы может служить путевое UNIX-имя, на основании которого формируется имя гнезда для домена типа UNIX. Им может быть также номер порта и хост-имя (необязательно), используемые при создании гнезда для домена типа Internet. Если в последнем случае хост-имя не указано, то используется хост-имя локальной машины. Отметим, что в ситуации, когда с помощью *sock::bind* создается гнездо домена Internet, названная функция выводит значение номера порта в стандартный поток ошибок. Это делается для того, чтобы клиентский процесс мог обращаться к порту с этим номером при создании гнезда, предназначенного для взаимодействия с гнездом сервера.

После того как гнездо создано и получило имя, серверный процесс ожидает установления клиентского соединения через это гнездо. Затем с помощью функции *sock::read* он принимает сообщение MSG1 из клиентского гнезда, выводит это сообщение на экран и посредством функции *sock::write* передает в клиентский процесс сообщение MSG2. Сервер читает сообщение MSG2, поступившее из клиентского гнезда, с помощью функции *sock::readfrom* и отвечает клиенту сообщением MSG4, передаваемым с использованием API *write*. Наконец, серверный процесс завершается и гнездо уничтожается посредством функции-деструктора *sock::~sock*.

Ниже приведен текст программы-клиента, которая взаимодействует с указанным сервером:

```
#include "sock.h"

const char* MSG1 = "Hello MSG1";
const char* MSG3 = "Hello MSG3";

main( int argc, char* argv(])
{
   if (argc < 2) {
      cerr << "usage: " << argv(0] << " <sockname|port> [<host>]\n";
      return 1;
   }

   int  port=-1, rc;

   /* проверить, указан ли номер порта гнезда */
   (void)sscanf(argv[1],"%d",&port);

   /* host может быть именем гнезда или хост-именем */
   char buf[80], *host= (port==-1) ? argv[1] : argv[2], socknm[80];
```

```
   /* создать клиентское гнездо */
   sock sp( port!=-1 ? AF_INET : AF_UNIX, SOCK_STREAM );
   if (!sp.good()) return 1;

   /* соединиться с серверным гнездом */
   if (sp.connect(host,port) < 0) return 8;

   /* передать сообщение MSG1 на сервер */
   if (sp.write(MSG1, strlen(MSG1)+1) < 0) return 9;

   /* прочитать сообщение MSG2 с сервера */
   if (sp.read(buf,sizeof buf) < 0) return 10;
   cerr << "client: recv '" << buf << "'\n";

   /* передать сообщение MSG3 на сервер */
   if ((rc=sp.writeto( MSG3, strlen(MSG3)+1, 0, host, port, -1))
        < 0) return 11;

   /* прочитать сообщение MSG4 с сервера */
   if ((rc=read(sp.fd(),buf,sizeof buf))==-1) return 12;
   cerr << "client: read msg: " << buf << endl;

   /* закрыть гнездо */
   sp.shutdown();
}
```

Аргументы командной строки для этой программы — те же, что и в программе-сервере: путевое UNIX-имя или имя порта и необязательное хост-имя. Пользуясь этими аргументами, программа создает гнездо домена UNIX или гнездо домена Internet.

Для соединения с серверным гнездом клиентская программа вызывает функцию *sock::connect*. С помощью функции *sock::write* она посылает серверу сообщение MSG1, а затем читает сообщение MSG2 (посредством вызова функции *sock::read*). После этого клиент посылает на сервер с помощью функции *sock::writeto* сообщение MSG3 и читает ответное сообщение MSG4, пользуясь для этого API *read*. Когда клиентский процесс завершается, он удаляет гнездо посредством вызова функции *sock::shutdown*().

Эта система типа клиент/сервер может работать с гнездами домена UNIX или с гнездами домена Internet. Ниже приведен пример взаимодействия между серверным и клиентским процессами, осуществляемого при помощи гнезд домена UNIX. Имя серверного гнезда произвольно устанавливается как SOCK:

```
% CC -o sock_stream_srv sock_stream_srv.C -lsocket -lnsl
% CC -o sock_stream_cls sock_stream_cls.C -lsocket -lnsl
% sock_stream_srv SOCK &
[1]  373
% sock_stream_cls SOCK
server: receive msg: 'Hello MSG1'
client: recv 'Hello MSG2'
```



```
server: recvfrom msg: 'Hello MSG3'
client: read msg: 'Hello MSG4'
[1] + Done   sock_stream_srv SOCK
```

Отметим, что когда сервер посредством вызова функции *sock::readfrom* получает от клиента сообщение MSG3, в переменную *socknm* заносится значение NULL. Клиентскому гнезду имя не присваивается. В конфигурации клиент/сервер, как правило, имя присваивается только серверному гнезду. Это позволяет клиентским гнездам устанавливать с ним соединения, но не наоборот.

Приведенный ниже пример показывает, как осуществляется взаимодействие между серверным и клиентским процессами при помощи гнезд домена Internet. Здесь имя машины, на которой работают оба процесса, — *fruit*. Поскольку имя машины указано, система может сама выбрать свободные номера портов для клиентского и серверного гнезд:

```
% sock_stream_srv 0 fruit &
[1] 374
Socket port: 32804
% sock_stream_cls 32804 fruit
server: receive msg: 'Hello MSG1'
client: recv 'Hello MSG2'
server: recvfrom 'fruit' msg: 'Hello MSG3'
client: read msg: 'Hello MSG4'
[1] + Done   sock_stream_srv 0 fruit
```

Отметим, что здесь выполняются те же самые программы типа клиент/сервер, но с разными аргументами командной строки. Процессы, взаимодействующие по схеме клиент/сервер, выполняются с использованием гнезд домена Internet. Результаты выполнения этих программ такие же, как и в случае с гнездами домена UNIX. Более того, хотя в приведенном выше примере серверный и клиентский процессы выполняются на одной машине, при работе на разных машинах результат был бы точно таким же. Разница заключается лишь в том, что серверная программа выполняется на одной машине (например, fruit); а клиентская программа выполняется на удаленной машине с указанием номера порта серверного гнезда и хост-имени как аргументов командной строки. Сообщения, посылаемые сервером (т.е. сообщения MSG1 и MSG3) отображаются на его хост-машине, а сообщения, посылаемые клиентами (т.е. MSG2 и MSG4) — на их удаленной хост-машине.

В программах с дейтаграммными гнездами тоже используется заголовок *sock.h*. Первая программа, *sock_datagram_srv.C*, выглядит так:

```
#include "sock.h"
const char* MSG2 = "Hello MSG2";
const char* MSG4 = "Hello MSG4";

main( int argc, char* argv[])
{
   char buf[80], socknm[80];
   if (argc < 2) {
```

```
   cerr << "usage: " << argv[0] << " <sockname|port>
        [<remote-host>]\n";
   return 1;
}
int port = -1, rc;

/* проверить, указан ли номер порта или имя гнезда */
(void)sscanf( argv[1],"%d",&port);

/* создать дейтаграммное гнездо */
sock sp( port==-1 ? AF_UNIX : AF_INET, SOCK_DGRAM );
if (!sp.good()) return 1;

/* присвоить имя гнезду */
if (sp.bind(port==-1 ? argv[1] : argv[2],port) < 0) return 2;

/* прочитать сообщение MSG1 от клиента */
if ((rc=sp.readfrom( buf, sizeof buf, 0, socknm, &port, -1))
     < 0) return 1;
cerr << "server: recvfrom from '" << socknm << "' msg: " << buf
     << endl;

/* направить сообщение MSG2 клиенту */
if ((rc= sp.writeto( MSG2, strlen(MSG2)+1, 0, socknm, port, -1))
      < 0) return 2;

/* установить адрес клиента, принимаемый по умолчанию */
if ((rc = sp.connect(socknm, port)) < 0) return 3;

/* прочитать сообщение MSG3 от клиента */
if ((rc = sp.read(buf, sizeof buf, 0)) < 0) return 4;
cerr << "server: receive msg: '" << buf << "'\n";

/* направить сообщение MSG4 клиенту */
if (write(sp.fd(),MSG4,strlen(MSG4)+1)<0) return 5;
}
```

Эта программа похожа на программу *sock_stream_srv.C*. Различаются они лишь тем, что создаваемое гнездо объявляется как SOCK_DGRAM (это делается посредством функции *sock::sock*). Программа получает в командной строке имя гнезда (необходимо для создания гнезда домена UNIX) или номер порта и/или имя хоста (для создания гнезда домена Internet). Имя гнезда должно быть известно клиенту, желающему установить с ним соединение. После того как гнездо будет создано, программа начнет читать сообщение клиента, пользуясь функцией *sock::readfrom*, которая возвращает имя гнезда клиента. Программа отвечает клиенту, посылая ему сообщение MSG2 с помощью функции *sock::writeto*. После этого программа посредством вызова *sock::connect* устанавливает соединение с клиентом, применяя при этом адрес, заданный по умолчанию. Затем для чтения сообщения MSG3 используется

функция *sock::read* и, в заключение, с помощью API *write* посылается сообщение MSG4.

Клиентская программа *sock_datagram_cls.C*, взаимодействующая с только что рассмотренной программой, приведена ниже:

```
#include "sock.h"
const char* MSG1 = "Hello MSG1";
const char* MSG3 = "Hello MSG3";

main( int argc, char* argv[])
{
   char buf[80], socknm[80];
   if (argc < 2) {
      cerr << "usage: " << argv[0]
           << "  me|port> [ <remote-host>]\n";
      return 1;
   }
   int  nlen, port = -1, rc;

   /* проверить, указан ли номер порта или имя гнезда */
   (void)sscanf(argv[1],"%d",&port);

   /* создать гнездо с использованием дейтаграмм */
   sock sp( port==-1 ? AF_UNIX : AF_INET, SOCK_DGRAM );
   if (!sp.good()) return 1;

   if (port==-1) {                        /* гнездо домена UNIX */
      sprintf(buf,"%s%d", argv[1], getpid()); /* создание имени
                                                 клиентского гнезда */
      if (sp.bind(buf,port) < 0) return 2;  /* присвоение имени
                                               гнезду */
   } else
      if (sp.bind(0,0) < 0) return 2; // присвоение имени гнезду

   /* запись сообщения MSG серверу */
   if ((rc=sp.writeto( MSG1, strlen(MSG1)+1, 0,
        port==-1? argv[1] : argv[2], port, -1)) < 0) return 6;

   /* чтение сообщения MSG2, поступившего от сервера */
   if ((rc=sp.readfrom( buf, sizeof buf, 0, socknm, &port, -1))
        < 0) return 7;
   cerr << "client: recvfrom '" << socknm << "' msg: " << buf << endl;

   /* выяснить адрес сервера, заданный по умолчанию */
   if (sp.connect(socknm,port) < 0) return 8;

   /* направить сообщение MSG3 серверу  */
   if (sp.write(MSG3, strlen(MSG3)+1) < 0) return 9;
```

```
    /* чтение сообщения MSG4 сервера */
    if ((rc=read(sp.fd(),buf,sizeof buf))==-1) return 10;
    cerr << "client: read msg: " << buf << endl;
4980
    sp.shutdown();
}
```

В командной строке этой программы в качестве аргумента указывается имя гнезда клиента либо номер порта и/или имя хоста. Программа модифицирует имя гнезда (для домена UNIX) путем добавления к нему PID своего процесса. В случае использования гнезд домена Internet добавляется еще и номер порта. Создав гнездо (посредством вызова функций *sock::sock* и *sock::bind*), программа посылает серверу сообщение MSG1 с помощью *sock::writeto* и ожидает ответного сообщения MSG2, вызывая для этого функцию *sock::readfrom*. Затем программа выясняет адрес сервера, заданный по умолчанию (с помощью функции *sock::connect*). Для отправки сообщения MSG3 серверному процессу используются функция *sock::write* и API *read*. Эти функции позволяют также принять сообщение MSG4 от сервера. Перед завершением своей работы программа вызывает функцию *sock::shutdown* для удаления гнезда.

Пример работы этих программ с использованием гнезд домена UNIX имеет следующий вид:

```
% CC -o sock_datagram_srv sock_datagram_srv.C -lsocket -lnsl
% CC -o sock_datagram_cls sock_datagram_cls.C -lsocket -lnsl
% sock_datagram_srv SOCK_DG &
% sock_datagram_cls SOCK_DG
server: recvfrom from 'SOCK_DG572' msg: Hello MSG1
client: recvfrom 'SOCK_DG' msg: Hello MSG2
server: receive msg: Hello MSG3
client: read msg: Hello MSG4
[1] + Done    sock_datagram_srv SOCK_DG
```

Отметим, что в командной строке имя гнезда было задано как SOCK_DG, в то время как полное его имя — SOCK_DG572. Выходная информация этих программ сходна с результатами работы программ, в которых используются потоковые гнезда.

В случае создания гнезд домена Internet результат работы этих же программ будет следующим:

```
% sock_datagram_srv 0 fruit &
Socket port: 32838
% sock_datagram_cls 32838 fruit
Socket port: 32840
server: recvfrom from 'fruit' msg: Hello MSG1
client: recvfrom 'fruit' msg: Hello MSG2
server: receive msg: Hello MSG3
client: read msg: Hello MSG4
[1] + Done    sock_datagram_srv 0 fruit
```


В этом примере номер порта, используемого первым процессом, равен 32838. Он выполняется на машине с именем *fruit*. Номер порта второго процесса — 32840. Он также выполняется на машине *fruit*. Оба процесса взаимодействуют точно так же, как и в случае гнезд домена UNIX. Выходная информация программ, за исключением номеров портов, совпадает.

11.3. Пример приложения типа клиент/сервер, предназначенного для обработки сообщений

В этом разделе представлена новая версия приложения типа клиент/сервер, рассмотренного в разделе 10.3.7. В этой версии используются потоковые гнезда, с помощью которых организуется канал связи между сервером сообщений и любым из его клиентских процессов. Поскольку сервер соединяется непосредственно с каждым клиентским процессом, всякое посылаемое клиентом сообщение является, по сути, представленной в виде строки символов командой, которую необходимо выполнить (например, LOCAL_TIME, GMT_TIME, QWUIT_CMD и др.). Сервер посылает свой ответ клиенту тоже в виде строки символов.

Ниже приведена программа сервера сообщений *sock_msg_srv.C*:

```
#include "sock.h"
#include <sys/times.h>
#include <sys/types.h>

#define MSG1 "Invalid cmd to message server"

typedef enum { LOCAL_TIME, GMT_TIME, QUIT_CMD, ILLEGAL_CMD } CMDS;

/* обработать команды клиента */
int process_cmd (int fd )
{
   char    buf[80];
   time_t  tim;
   char*   cptr;

   /* читать команды клиента до символов EOF или QUIT_CMD */
   while (read(fd, buf, sizeof buf) > 0)
   {
      int     cmd = ILLEGAL_CMD;
      (void)sscanf(buf,"%d",&cmd);
      switch (cmd) {
         case LOCAL_TIME:
            tim = time(0);
            cptr = ctime(&tim);
            write(fd, cptr, strlen(cptr)+1);
            break;
```

```
        case GMT_TIME:
           tim = time(0);
           cptr = asctime(gmtime(&tim));
           write(fd, cptr, strlen(cptr)+1);
           break;
        case QUIT_CMD:
           return cmd;
        default:
           write(fd, MSG1, sizeof MSG1);
     }
   }
   return 0;
}

int main( int argc, char* argv[])
{
   char  buf[80], socknm[80];
   int  port=-1, nsid, rc;
   fd_set  select_set;
   struct timeval timeRec;

   if (argc < 2) {
      cerr << "usage: " << argv[0] << " <sockname|port> [<host>]\n";
      return 1;
   }

   /* проверить, указан ли номер порта в имени гнезда */
   (void)sscanf(argv[1],"%d",&port);

   /* создать потоковое гнездо */
   sock sp( port!=-1 ? AF_INET : AF_UNIX, SOCK_STREAM );
   if (!sp.good()) return 1;

   /* присвоить имя серверному гнезду */
   if (sp.bind(port==-1 ? argv[1] : argv[2],port) < 0) return 2;

   for (;;)   // проверка наличия клиентских соединений
   {
      timeRec.tv_sec = 1;  // задает тайм-аут опроса равным 1 секунде
      timeRec.tv_usec= 0;
      FD_ZERO( &select_set );
      FD_SET( sp.fd(), &select_set );

      /* ожидать наступления тайм-аута или начала процедуры чтения
         для гнезда */
      rc = select(FD_SETSIZE, &select_set, 0, 0, &timeRec );
      if (rc > 0 && FD_ISSET(sp.fd(), &select_set))
      {
         /* принять из клиентского гнезда запрос на соединение */
         if ((nsid = sp.accept(0, 0)) < 0) return 1;
         /* обработать команды */
         if (process_cmd(nsid)==QUIT_CMD) break;
```



```
         close(nsid); /* повторно использовать дескриптор файла */
      }
      /* в противном случае выполнять какие-либо другие действия */
   }
   sp.shutdown();
   return 0;
}
```

Синтаксис вызова сервера сообщений такой же, как и в случае вызова программы *sock_stream_srv.C*. Создается потоковое гнездо, которому точно так же присваивается имя. Однако после того, как гнездо организовано, сервер с помощью API *select* проверяет, имеются ли в нем запросы на обслуживание. Проверка осуществляется с интервалом, равным 1 секунде, чтобы в перерывах между проверками сервер мог выполнять другие операции.

Когда клиент посылает серверу команду на обслуживание, он вызывает функцию *process_cmd*, с помощью которой обрабатываются все команды на обслуживание, инициированные клиентом. Функция *process_cmd* возвращает результат, когда клиент посылает серверу команду QUIT_CMD. Сервер освобождает дескриптор файла *nsid*, который обозначает гнездо, созданное вызовом функции *sock::accept*. После этого сервер либо продолжает опрашивать потоковое гнездо на предмет соединения с другим клиентом, либо просто удаляет его и завершается.

Программа-клиент *sock_msg_cls.C* выглядит следующим образом:

```
#include "sock.h"
#define QUIT_CMD 2

int main( int argc, char* argv[])
{
   if (argc < 2) {
      cerr << "usage: " << argv[0] << " <sockname|port> [<host>]\n";
      return 1;
   }

   int  port=-1, rc;

   /* проверить, указан ли номер порта */
   (void)sscanf(argv[1],"%d",&port);

   /* host может быть именем гнезда или хост-именем */
   char buf[80], *host= (port==-1) ? argv[1] : argv[2], socknm[80];

   /* создать клиентское гнездо */
   sock sp( port!=-1 ? AF_INET : AF_UNIX, SOCK_STREAM );
   if (!sp.good()) return 1;

   /* установить соединение с серверным гнездом */
   if (sp.connect(host,port) < 0) return 8;
```

```
    /* передать команды 0, 1, 2 серверу */
    for (int cmd=0; cmd < 3; cmd++)
    {
       /* сформировать команду для сервера */
       sprintf(buf,"%d",cmd);
       if (sp.write(buf,strlen(buf)+1) < 0) return 9;

       /* выйти из цикла, если это команда QUIT_CMD */
       if (cmd==QUIT_CMD) break;

       /* прочитать ответ сервера */
       if (sp.read(buf,sizeof buf) < 0) return 10;
       cerr << "client: recv '" << buf << "'\n";
    }
    sp.shutdown();
    return 0;
}
```

Синтаксис этой программы-клиента такой же, как в примере программы *sock_stream.cls.C*. Она точно так же создает потоковое гнездо и соединяет его с серверным гнездом. После организации гнезда клиент передает серверу следующие запросы на обслуживание: сообщить дату и местное время; сообщить дату и время по Гринвичу; выполнить команду завершения работы сервера. Затем клиент читает завершающее сообщение сервера.

После отправки каждой команды (кроме QUIT_CMD) на обслуживание клиент принимает ответ сервера и направляет результаты на стандартный вывод. Завершает свою работу программа-клиент после того, как пошлет серверу команду QUIT_CMD.

Программа-клиент и программа-сервер компилируются следующим образом:

```
% CC -o sock_msg_srv sock_msg_srv.C -lsocket -lnsl
% CC -o sock_msg_cls sock_msg_cls.C -lsocket -lnsl
```

Вот примерный протокол взаимодействия этих программ:

```
% sock_msg_srv 0 fruit &
[1] 441
Socket port: 32792
% sock_msg_cls 32792 fruit
client: recv 'Sun Feb 12 00:41:25 1997'
client: recv 'Sun Feb 12 08:41:25 1997'
[1] + Done    sock_msg_srv 0 fruit
```

В приведенном выше примере клиентский и серверный процессы взаимодействуют друг с другом с помощью потоковых гнезд домена Internet. Клиент читает только те сообщения сервера, которые тот посылает в ответ на команды LOCAL_TIME и UTC_TIME. Процессы, использующие гнезда домена Internet, могут работать и на отдельных машинах.

В следующем примере показано, как те же самые процессы взаимодействуют с помощью гнезд домена UNIX:

```
% sock_msg_srv SOCK_MSG &
[1] 446
% sock_msg_cls SOCK_MSG
client: recv 'Sun Feb 12 00:42:38 1997'
client: recv 'Sun Feb 12 08:42:38 1997'
[1] + Done    sock_msg_srv SOCK_MSG
```

11.4. Интерфейс транспортного уровня (TLI)

Интерфейс транспортного уровня (Transport Level Interface, TLI) был разработан в рамках ОС UNIX System V.3 в качестве альтернативы гнездам. TLI более гибок, чем гнезда, и построен на основе механизма STREAMS, поддерживающего большинство транспортных протоколов. TLI создает конечные точки транспортировки (transport endpoints), поведение и функции которых похожи на поведение и функции гнезд. Например, эти конечные точки транспортировки могут взаимодействовать друг с другом как в режиме с установлением соединения, так и в режиме без установления соединения. Кроме того, процессы, работающие на разных машинах или на одной машине, могут общаться между собой, используя свои конечные точки транспортировки. И гнезда, и конечные точки TLI обозначаются дескрипторами. Процесс может установить для этих дескрипторов флаг O_NONBLOCK либо при их назначении, либо при вызове функции *fcntl*. Это приводит к тому, что соответствующие операции с гнездами и конечными точками TLI выполняются в неблокирующем режиме.

Конечная точка TLI не может взаимодействовать с гнездом. Когда такая точка создается, пользователь должен указать транспортный протокол, который с ней должен быть связан. При создании гнезда указывать транспортный протокол пользователю не приходится. API *socket* выбирает протокол по умолчанию на основании типа гнезда. Помимо этих различий, есть еще одно: адрес, назначаемый гнезду для внутримашинной связи, отличается от соответствующего адреса конечной точки TLI. Конечной точке TLI для связи с другими конечными точками транспортировки присваивается целочисленный номер порта, тогда как гнезду для работы в таком режиме присваивается путевое UNIX-имя.

В ходе проведения сеанса связи по Internet конечной точке TLI присваивается хост-имя и номер порта (аналогично тому, как это делается для гнезда). При связи с установлением соединения адреса клиентским гнездам присваиваются только в том случае, если клиентские процессы назначают эти адреса явно. Конечным точкам транспортировки адреса присваиваются всегда — либо пользователями, либо базовым транспортным протоколом.

Между API TLI и API гнезд существует практически однозначное соответствие. Это облегчает конвертирование приложений, ориентированных на использование гнезд, в TLI-приложения. В следующем разделе дается обзор

интерфейсов прикладного программирования TLI и проводится их сравнение с API гнезд. В последующих разделах более подробно описываются синтаксис и методика использования этих API. В последних двух разделах приведены два примера TLI-приложений. Одно из них является вариантом представленной в разделе 11.2 программы типа клиент/сервер, в которой вместо гнезд применяются конечные точки TLI. Во втором примере показано, как с помощью конечных точек транспортировки можно посылать дейтаграммные сообщения.

11.4.1. API интерфейса транспортного уровня

Системные функции TLI и их назначение описаны ниже.

API TLI	Назначение
t_open	Создает конечную точку транспортировки и задает базовый транспортный протокол
t_bind	Присваивает имя конечной точке транспортировки. Для конечной точки, ориентированной на установление соединения, указывается также максимально допустимое количество запросов на соединение
t_listen	Ожидает запроса на соединение от клиентской конечной точки транспортировки
t_accept	Принимает запрос на соединение от клиентской конечной точки транспортировки
t_connect	Посылает запрос на соединение в серверную конечную точку транспортировки
t_snd	Посылает сообщение в конечную точку, с которой уже установлено соединение. Применяется только для конечной точки транспортировки, ориентированной на соединение
t_rcv	Принимает сообщение из конечной точки, с которой уже установлено соединение. Применяется только для конечной точки транспортировки, ориентированной на соединение
t_sndudata	Посылает дейтаграммное сообщение в конечную точку транспортировки с заданным адресом
t_rcvudata	Принимает из конечной точки транспортировки дейтаграммное сообщение и адрес отправителя
t_snddis	Разрывает соединение
t_rcvdis	Возвращает признак разрыва соединения и сообщает, почему оно было разорвано
t_sndrel	Посылает в конечную точку транспортировки запрос на прекращение передачи через эту точку
t_rcvrel	Возвращает из конечной точки транспортировки, с которой установлено соединение, признак того, что прием сообщений через эту точку осуществляться не будет
t_error	Действует аналогично функции *perror*. В случае неудачного завершения вызова TLI-функции выводит на экран сообщение об ошибке

API TLI	Назначение
t_alloc	Выделяет динамическую память для конечной точки транспортировки
t_free	Освобождает динамическую память, выделенную конечной точке транспортировки
t_close	Освобождает дескриптор конечной точки транспортировки

Последовательность вызова описанных API TLI при установлении соединения между сервером и клиентом с использованием виртуального канала представлена на рис. 11.3.

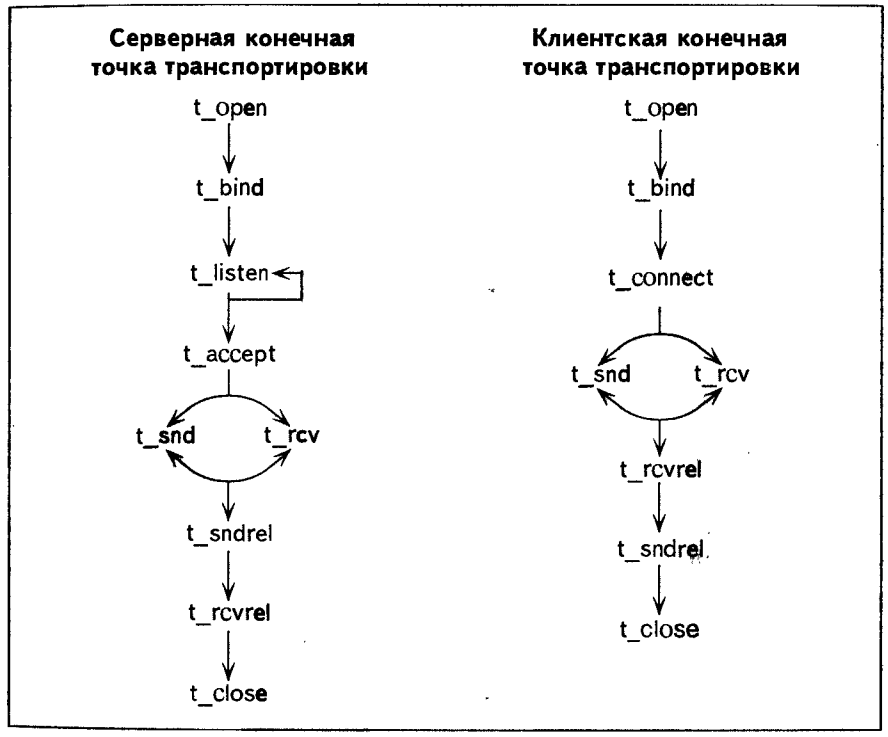


Рис. 11.3. Последовательность вызовов API TLI для серверной и клиентской конечных точек транспортировки

Обратите внимание на сходство этих последовательностей с теми, которые изображены на рис. 11.1. В частности, API *t_open* похож на API *socket*, API *t_bind* — на API *bind*, а функции *t_snd* и *t_rcv* — на API *send* и *rcv*. При этом, однако, API *t_listen* — не то же самое, что API *listen*. API *listen* задает максимально допустимое для гнезда число ожидающих своей очереди запросов на соединение. Для конечной точки транспортировки эта информация задается в API *t_bind*. API *t_listen* очень похож на API гнезд *accept*: он заставляет процесс ждать клиентского запроса на соединение. После получения

такого запроса функция *t_listen* возвращает адрес клиентской конечной точки транспортировки. Сервер может вызвать либо функцию *t_accept* — для установления соединения, либо функцию *t_snddis* — для разрыва соединения.

Как и API *accept*, API *t_accept* назначает конкретной конечной точке транспортировки дескриптор, необходимый серверу для установления связи с клиентским процессом. Серверный процесс может продолжать мониторинг последующих запросов на соединение, поступающих из клиентских процессов, используя для этого дескриптор, полученный в результате вызова функции *t_open*.

Когда серверный и клиентский процессы заканчивают взаимодействие, один из них может вызвать функцию *t_sndrel*, которая пошлет процессу-партнеру уведомление о разъединении. Этот партнер вызовет функцию *t_rcvrel* для приема уведомления и ответит посредством вызова функции *t_sndrel*. После получения первым процессом ответного уведомления (в результате вызова функции *t_rvcrel*) соединение между конечными точками транспортировки будет разорвано и оба процесса смогут вызвать функцию *t_close*, чтобы удалить свои конечные точки транспортировки.

В качестве альтернативы функциям *t_sndrel* и *t_rcvrel* серверный и клиентский процессы могут вызвать для разрыва соединения функции *t_snddis* и *t_rcvdis*. Функция *t_snddis* используется для аварийного разрыва соединения: все данные, еще не переданные через конечные точки транспортировки, немедленно уничтожаются. Функция *t_sndrel* используется для неаварийного разрыва соединения: все неотправленные данные доставляются по назначению и лишь после этого соединение разрывается. Все транспортные протоколы, используемые с TLI, должны поддерживать функции *t_snddis* и *t_rcvdis*, тогда как требование о поддержке функций *t_sndrel* и *t_rcvrel* не является обязательным.

Последовательность вызовов API TLI, предназначенных для создания дейтграммной конечной точки транспортировки, приведена на рис. 11.4.

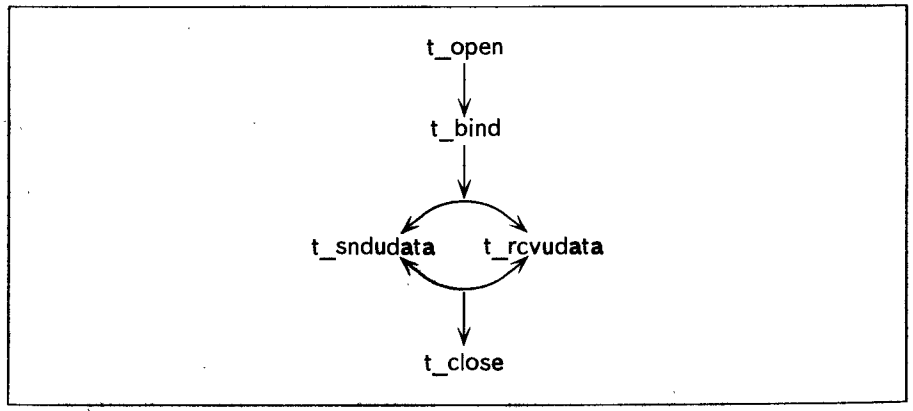


Рис. 11.4. Последовательность вызовов API TLI для создания дейтаграммной конечной точки транспортировки

Манипулировать дейтаграммными конечными точками транспортировки не трудно: процесс вызывает функцию *t_open* — для создания конечной точки, а затем функцию *t_bind* — для присвоения ей имени. После этого процесс вызывает функцию *t_sndudata*, чтобы передать сообщения другим процессам. Причем каждое сообщение снабжается адресом конечной точки отправителя. Процесс может также получать сообщения от других процессов, для чего вызывается функция *t_rcvudata*. Каждое принятое сообщение снабжается адресом конечной точки отправителя, чтобы процесс мог отправить ему ответ.

После того как процесс завершает свое участие в межпроцессном взаимодействии, он вызывает функцию *t_close*, которая освобождает дескриптор конечной точки транспортировки. Функцию *t_snddis* или *t_sndrel* здесь вызывать не нужно, поскольку виртуальный канал для связи с другими процессами не создается.

В следующих разделах даются более детальные пояснения по синтаксису API TLI и их использованию.

11.4.2. Функция t_open

Прототип функции API *t_open* выглядит следующим образом:

```
#include <tiuser.h>
#include <fcntl.h>

int t_open (char* path, int aflag, struct t_info* info);
```

Эта функция создает конечную точку транспортировки, которая использует провайдер транспорта*, указанный в аргументе *path*. Фактическим значением этого аргумента может быть путевое имя файла устройства, соответствующего используемому провайдеру транспорта. Например, файл */dev/ticlts* соответствует провайдеру, функционирующему на основе протокола UDP, а файл */dev/ticotsord* — провайдеру транспорта, использующему виртуальный канал.

Аргумент *aflag* задает режим доступа к конечной точке транспортировки для вызывающего процесса. Его значение, определяемое в заголовке <fcntl.h>, обычно равно O_RDWR. Это означает, что вызывающий процесс может передавать и принимать сообщения через конечную точку транспортировки. Вместе с этим флагом может указываться флаг O_NONBLOCK, который позволяет конечной точке транспортировки выполнять неблокирующие операции.

* Провайдер транспорта — транспортный протокол, поддерживающий функционирование прикладных программ и протоколов высокого уровня и обеспечивающий их независимость от физической среды передачи и протоколов более низких уровней. — *Прим. ред.*

Аргумент *info* возвращает стандартные характеристики используемого провайдера транспорта. Эта информация обычно игнорируется, и фактическое значение *info* может быть равно 0. Если же пользователь хочет проверить стандартные характеристики, он должен определить переменную типа *struct t_info* и передать ее адрес в аргумент *info*. После того как рассматриваемая функция возвратит управление вызывающему процессу, пользователь сможет увидеть содержимое этой переменной. Интересующие его данные содержатся в поле *info->servtype*, которое может иметь одно из перечисленных ниже значений.

Значение *servtype*	Смысл
T_COTS	Провайдер транспорта позволяет устанавливать соединение на основе виртуального канала, но не поддерживает функцию прекращения передачи сообщений только в одном из направлений
T_COTSORD	Провайдер транспорта позволяет устанавливать соединение на основе виртуального канала и поддерживает функцию прекращения передачи сообщений только в одном из направлений
T_CLTS	Провайдер транспорта поддерживает передачу дейтаграммных сообщений

Эти константы объявляются в заголовке <tiuser.h>. В случае неудачи рассматриваемая функция возвращает -1, а в случае успешного выполнения — дескриптор, который обозначает конечную точку транспортировки, созданную функцией.

Приведенные ниже операторы предназначены для создания ориентированной на соединение конечной точки транспортировки, которая поддерживает функцию прекращения передачи сообщений только в одном из направлений. Операции в этой конечной точке выполняются в неблокирующем режиме. Информация о стандартных характеристиках провайдера транспорта не запрашивается. Дескриптор конечной точки транспортировки присваивается переменной *fd*:

```
int fd = t_open("/dev/ticotsord", O_RDWR | O_NONBLOCK, 0)
if (fd == -1) t_error("t_open");
```

Рассмотрим операторы, предназначенные для создания дейтаграммной конечной точки транспортировки. Операции в этой конечной точке выполняются в блокирующем режиме. Информация о стандартных характеристиках провайдера транспорта возвращается в переменной *info*. Значение дескриптора конечной точки транспортировки присваивается переменной *fd*:

```
struct t_info info;
int fd = t_open("/dev/ticlts", O_RDWR, &info)
if (fd == -1) t_error("t_open");
```


Наиболее часто используемые конечные точки транспортировки имеют заранее определенные адреса, указанные в файле */etc/services*. Например, следующие две записи этого файла определяют две конечные точки:

```
# /etc/services
test    4045/tcp
utst1   5001/udp
```

Здесь имя сервисной программы, использующей первую конечную точку транспортировки,— *test*, а в качестве провайдера транспорта используется протокол *TCP*. Таким образом, это — конечная точка, ориентированная на соединение. Имя сервиса второй конечной точки транспортировки — *utst1*, а в качестве провайдера транспорта используется протокол *UDP*. Следовательно, это — дейтаграммная конечная точка.

При наличии в файле */etc/services* таких определений адрес конечной точки транспортировки и имя файла устройства ее провайдера транспорта можно задать следующим образом:

```
struct nd_hostserv hostserv;
struct netconfig   *nconf;
struct nd_addrlist *addr;
void               *hp;
int                type = NC_TPI_COTS_ORD;
if ((hp=setnetpath()) == 0)
{
   perror("Can't init network");
   exit(1);
}
hostserv.h_host = "fruit"   // задается хост-имя машины
hostserv.h_serv = "test"    // задается имя сервисной программы
while ((nconf=getnetpath(hp)) != 0)
{
   if (nconf->nc_semantics == type
      && netdir_getbyname(nconf, &hostserv, &addr)==0)
      break;
}
endnetpath(hp);

if (nconf == 0)
   cerr << "No transport found for service: "test"\n";
else if ((tid=t_open(nconf->nc_device, O_RDWR, 0) < 0)
    t_error("t_open fails");
else cerr << "transport end point's address is specified in addr\n";
```

В приведенном сегменте кода имя машины и имя сервисной программы задаются как *fruit* и *test*. Эта информация заносится в переменную *hostserv*. Функции *setnetpath*, *getnetpath* и *endnetpath* используются для обработки записей файла */etc/netconfig*, каждая из которых содержит имя провайдера транспорта и имя соответствующего файла устройства. В этом примере программа ищет провайдер транспорта типа NC_TPI_COTS_ORD (с установлением соединения и поддержкой разрыва соединения только в одном

из направлений). Для каждого провайдера транспорта, тип которого удовлетворяет этим критериям, в функцию *netdir_getbyname* передаются переменные *nconf* и *hostserv*. Это позволяет найти, во-первых, адрес конечной точки транспортировки на указанной машине (в нашем примере — *fruit*) и, во-вторых, имя заданной сервисной программы (*test*).

Адрес конечной точки транспортировки возвращается с помощью переменной *addr*. Эта переменная используется позже в вызовах функций *t_bind* (серверный процесс) и *t_connect* (клиентский процесс).

Приведенный выше код можно модифицировать для получения адреса дейтаграммной конечной точки транспортировки. Этот адрес является путевым именем файла устройства, соответствующего заданному для сервиса *utst1* провайдеру транспорта. Модификация предусматривает следующие операции:

- присвоить переменной *type* значение NC_TPI_CLTS вместо NC_TPI_COTS_ORD;
- указать в поле *hostserv.h_serv* значение *utst1*.

11.4.3. Функция t_bind

Прототип функции *t_bind* выглядит следующим образом:

```
#include <tiuser.h>

int t_bind ( int fd, struct t_bind* inaddr, struct t_bind* outaddr );
```

С помощью этой функции конечной точке транспортировки присваивается имя (или адрес). Дескриптор этой конечной точки указывается аргументом *fd*. Фактическое значение *fd* берется из вызова *t_open*.

Аргумент *inaddr* содержит адрес, присвоенный конечной точке транспортировки. Его фактическое значение может быть NULL. Это означает, что адрес конечной точке должен присвоить провайдер транспорта.

Структура *t_bind* объявляется следующим образом:

```
struct t_bind
{
  struct netbuf   addr;
  unsigned        qlen;
};
```

Здесь поле *qlen* задает максимально допустимое для данной конечной точки транспортировки количество запросов на соединение. Применительно к серверной конечной точке транспортировки этому полю присваивается ненулевое значение, а для клиентских точек — нулевое. Поле *addr* содержит адрес, который должен быть присвоен конечной точке транспортировки.

Структура *netbuf* объявляется следующим образом:

```
struct netbuf
{
  unsigned int  maxlen;
  unsigned int  len;
  char*         buf;
};
```

Здесь значение *len* задает число символов в аргументе *buf*, который содержит адрес конечной точки транспортировки. Значение аргумента *maxlen* в данном случае не используется.

В аргументе *outaddr* возвращается фактический адрес, присвоенный конечной точке транспортировки используемым провайдером транспорта. Этот адрес может отличаться от того, который указан в *inaddr*. Если провайдер транспорта не может назначить конечной точке адрес, указанный в *inaddr*, он назначает ей другой адрес. Фактическое значение аргумента *outaddr* может быть равно NULL. Это означает, что вызывающему процессу безразлично, какой адрес назначается данной конечной точке транспортировки. Такая ситуация обычно характерна для клиентских конечных точек транспортировки, функционирующих в режиме с установлением соединения.

Если значение аргумента *outaddr* — адрес переменной типа *struct t_bind*, то *outaddr->buf* является адресом буфера, определенного вызывающим процессом, а в поле *outaddr->maxlen* задается максимальный размер буфера *outaddr->buf*. При выходе из функции поле *outaddr->len* содержит число символов, имеющихся в *outaddr->buf*, где хранится адрес конечной точки транспортировки, назначенный провайдером транспорта.

В случае успешного выполнения эта функция возвращает 0, а в случае неудачи возвращает -1

Ниже конечной точке транспортировки, указанной аргументом *fd*, назначается адрес, возвращаемый функцией *netdir_getbyname* (см. пример из предыдущего раздела). Данная конечная точка одновременно может принимать до пяти клиентских запросов на соединение. Функция *t_alloc* динамически выделяет память объекту типа *struct t_bin* (на который указывает переменная *bind*). Это гарантирует правильную инициализацию всех полей названного объекта.

```
struct t_bind *bind = t_alloc(fd, T_BIND, T_ALL);
if (!bind)
    t_error("t_alloc fails for T_BIND");
else {
   bind->qlen = 5;
    bind->addr = *(addr->n_addrs);
   if (t_bind(fd, bind, bind) < 0) t_error("t_bind");
}
```

Конечной точке транспортировки можно присвоить адрес целочисленного типа. Это полезно, если данная точка взаимодействует с другими конечными точками транспортировки только на этой машине. В следующей

программе конечной точке транспортировки, указанной аргументом *fd*, присваивается адрес, значение которого равно 2:

```
struct t_bind *bind = t_alloc(fd, T_BIND, T_ALL);
if (bind)  {
    bind->qlen = 5;
    bind->addr.len = sizeof(int)
    *(int)bind->addr.buf = 2;;
    if (t_bind(fd, bind, bind) < 0) t_error("t_bind");
) else   t_error("t_alloc fails for T_BIND");
)
```

11.4.4. Функция t_listen

Прототип функции *t_listen* выглядит так:

```
#include <tiuser.h>

int t_listen ( int fd, struct t_call* call );
```

Эта функция ожидает прибытия в конечную точку транспортировки, указанную аргументом *fd*, клиентского запроса на соединение. Адрес клиентской конечной точки транспортировки возвращается посредством аргумента *call*.

По умолчанию эта функция блокирует вызывающий процесс до тех пор, пока не будет получен клиентский запрос на соединение. Если же точка *fd* задана как неблокирующая (с помощью флага O_NONBLOCK в вызове *t_call* или посредством функции *fcntl*), то *t_listen* возвращает управление немедленно (при условии, что ни одного клиентского запроса не обнаружено). Глобальной переменной *t_errno* присваивается значение TNODATA.

Структура *t_call* объявляется следующим образом:

```
struct t_call
{
  struct      addr;
  struct netbuf opt;
  struct netbuf udata;
  int           sequence;
};
```

Здесь поле *addr* содержит адрес клиентской конечной точки транспортировки, которая инициирует посылку запроса на соединение. В поле *opt* устанавливаются параметры, зависящие от протокола. В поле *udata* указываются необязательные пользовательские данные, которые должны быть посланы вместе с запросом на соединение. Поле *sequence* содержит целочисленный идентификатор, используемый для уникального обозначения каждого соединения.

В следующем примере проверяется наличие запроса на соединение в неблокирующем режиме. С помощью вызова функции *t_alloc* для переменной *call* динамически выделяется память. Такой способ выделения памяти гарантирует, что поля *maxlen* в аргументах *addr*, *opt* и *udata* будут отражать размеры соответствующих полей *buf*.

```
struct t_call *call = (struct t_call*)t_alloc(fd, T_CALLOC, T_ALL);
if (!call)
   t_error("t_alloc fails for T_CALL");
else do  (
   if (t_listen(fd, call) == 0) break;  /* получен запрос на
                                           соединение */
   if (t_errno != TNODATA) (
       t_error("t_listen fails");
       exit(1);
    )
   /* выполнять другие операции */
}  while(1);
```

В этом примере сначала вызывается функция *t_alloc*, которая динамически выделяет память под объект *struct t_call* и указывает на него переменной *call*. После этого начинается цикл, в котором вызывается функция *t_listen*, итеративно проверяющая наличие запросов на соединение. Возврат данной функцией нулевого значения говорит о том, что такой запрос имеется, и программа выходит из цикла. Когда *t_listen* возвращает ненулевое значение, проверяется глобальная переменная *t_errno*. Если ее значением является TNODATA, значит возникла какая-то иная ошибка. В таком случае программа вызывает функцию *t_error* для выдачи диагностического сообщения, а затем завершается. Какое-либо иное значение этой переменной говорит о том, что запрос на соединение еще не поступил, и программа выполняет другие операции, после чего возобновляет вызов *t_listen*.

11.4.5. Функция t_accept

Прототип функции *t_accept* выглядит следующим образом:

```
#include <tiuser.h>

int t_accept ( int fd, int newfd, struct t_call* call );
```

Эта функция принимает клиентский запрос на соединение, обнаруженный посредством вызова *t_listen*.

Аргумент *fd* указывает, какая серверная конечная точка транспортировки принимает запрос на соединение. Посредством аргумента *newfd* задается конечная точка, которая должна быть соединена с клиентской конечной точкой транспортировки. Значение *newfd* может как совпадать с *fd*, так и не совпадать. В первом случае в данной конечной точке транспортировки не

должно быть других ожидающих обработки запросов на соединение, иначе вызов функции завершится неудачей. Во втором случае дескриптор *newfd* должен быть определен посредством вызова *t_open* до использования в вызове данной функции. После успешного завершения вызова *t_accept* аргумент *fd* можно использовать для обработки запросов на соединение от других клиентов, а *newfd* применяется исключительно для связи с клиентом, адрес которого задан значением *call*. Это значение было получено в результате вызова функции *t_listen*.

В случае успешного выполнения данная функция возвращает 0, а в случае неудачи возвращает -1.

В следующем примере порождается процесс, который принимает запросы на соединение из одного или нескольких клиентских процессов. Обратите внимание на то, что конечная точка транспортировки, указанная аргументом *fd*, работает в блокирующем режиме:

```
struct t_call *call = (struct t_call*)t_alloc(fd, T_CALLOC, T_ALL);
if (!call)
  t_error("t_alloc fails for T_CALL");
else while (t_listen(fd, call) == 0)  /* получен один запрос на
                                         соединение */
   switch (fork())   {
      case -1: perror("fork"0; break;  /* родительский процесс; вызов
                                          fork завершился неудачно */
      default: break; /* родительский процесс; вызов fork выполнен
                         успешно */
      case 0:         /* порожденный процесс для связи с клиентом */
         if (newfd=t_open("/dev/ticotsord", O_RDWR, 0))==-1 ||
             t_bind(newfd,0,0)== -1)
             t_error("T_open or t_bind fails");
         else if (t_accept(fd, newfd, call)==-1)
             t_error("t_accept fails\n");
         else {
             t_close(fd);
            /* теперь — связь с клиентом посредством дескриптора newfd */
         }
   }
```

В этом примере сначала вызывается функция *t_alloc*, с помощью которой выделяется память под объект типа *struct t_call*, где будет храниться адрес клиента. Указателем на этот объект является переменная *call*. Если вызов *t_alloc* выполняется успешно, то начинается цикл, в котором вызывается функция *t_listen*, итеративно проверяющая наличие запросов на соединение. Для каждого принятого запроса программа создает порожденный процесс, обеспечивающий взаимодействие с соответствующим клиентом. В частности, каждый порожденный процесс создает новую конечную точку транспортировки того типа, который задан аргументом *fd*. Эту новую точку обозначает переменная *newfd*, и ей присваивается имя, назначенное провайдером транспорта. После этого порожденный процесс вызывает функцию *t_accept*, которая устанавливает соединение между конечной точкой, заданной

переменной *newfd*, и конечной точкой клиента. В случае успешного результата порожденный процесс освобождает свой экземпляр дескриптора *fd*, потому что он больше не нужен, и начинает взаимодействовать с клиентом.

11.4.6. Функция t_connect

Прототип функции *t_connect* выглядит следующим образом:

```
#include <tiuser.h>

int t_connect ( int fd, struct t_call* inaddr, struct t_call* outaddr );
```

Эта функция посылает в серверную конечную точку транспортировки запрос на соединение. Аргумент *fd* показывает, какая клиентская конечная точка транспортировки должна быть соединена с конечной точкой сервера. Адрес сервера задается аргументом *inaddr*, а адрес серверной конечной точки транспортировки — аргументом *outaddr*.

Значение аргумента *inaddr* не должно быть равным NULL. Значение аргумента *outaddr* может быть равным NULL, если адрес сервера не имеет значения.

По умолчанию рассматриваемая функция блокирует вызывающий процесс до тех пор, пока не будет установлено соединение с серверной конечной точкой транспортировки или не произойдет системная ошибка. Если же точка *fd* задана как неблокирующая, то данная функция инициирует выдачу запроса на установление соединения и возвращает управление немедленно (при условии, что соединение с серверной конечной точкой транспортировки не было установлено сразу же). Глобальной переменной *t_errno* присваивается значение TNODATA. Клиент может затем вызвать функцию *t_rcvconnect*, чтобы проверить успешность выполнения запроса на соединение. Прототип функции *t_rcvconnect* :

```
#include <tiuser.h>

int t_rcvconnect ( int fd, struct t_call* outaddr );
```

В случае успешного выполнения эта функция возвращает 0, а в случае неудачи возвращает -1.

В следующем примере конечная точка транспортировки переводится в неблокирующий режим работы, для чего используется вызов функции *fcntl*. Затем эта конечная точка пытается соединиться с сервером в неблокирующем режиме. Адрес сервера принимается равным 3.

```
struct t_call *call = (struct t_call*)t_alloc(fd, T_CALLOC, T_ALL);
if (!call) {
```

```
   t_error("t_alloc fails for T_CALL");
    exit(1);
)
call->addr.len = sizeof(int);      //задать адрес сервера
*(int)call->addr.buf = 3;
/* Установить fd как неблокирующий. Это можно сделать и при вызове
   t_open */
if (flg=fcntl(fd, F_GETFL, 0)==-1 ||
      fcntl(fd, F_SETFL, flg | O_NONBLOCK)==-1) (
   perror("fcntl");
   exit(2);
)
if (t_connect(fd, call, call))==-1) (
   while (t_errno == TNODATA) (      /* опросить на предмет завершения
                                        запроса на соединение */
/* выполнить другие операции */
      if (t_rcvconnect(fd, call)==0) break;
   }
   if (t_errno != TNODATA) {
      t_error("t_connect or t_rcvconnect fails");
      exit(4);
   }
} /* t_connect */
/* начать взаимодействие с серверной конечной точкой транспортировки */
```

В этом примере сначала вызывается функция *fcntl*, которая устанавливает неблокирующий режим работы для конечной точки транспортировки, указанной аргументом *fd*. Затем программа вызывает функцию *t_connect*, посредством которой устанавливается соединение с серверной конечной точкой транспортировки. Адрес сервера принимается равным 3 (только для локального соединения). Если *t_connect* завершается неудачно, программа входит в цикл, где выполняет другие операции, а затем вызывает функцию *t_rcvconnect* для проверки успешности запроса на соединение. Цикл завершается, когда *t_rcvconnect* возвращает код успешного выполнения (возвращаемое значение равно нулю) или когда возникает ошибка и *t_errno* не присваивается значение TNODATA. Если вызов функции *t_connect* или *t_rcvconnect* завершается успешно, программа начинает взаимодействие с серверным процессом.

11.4.7. Функции t_snd, t_sndudata

Прототипы функций *t_snd* и *t_sndudata* выглядят следующим образом:

```
#include <tiuser.h>

int t_snd ( int fd, char* buf, unsigned len, int flags );
int t_sndudata ( int fd, struct t_unitdata* udata );
```


Функция *t_snd* посылает сообщение длиной *len* байтов (которое содержится в буфере *buf*) в другой процесс, соединение с которым устанавливается через конечную точку транспортировки, указанную аргументом *fd*. Эта точка должна функционировать на основе виртуального канала и соединяться с другой конечной точкой посредством вызова *t_connect* (для клиентского процесса) или *t_accept* (для серверного процесса).

Аргумент *flags* может иметь нулевое (по умолчанию) или одно из указанных ниже значений, которые определяются в заголовке <tiuser.h>.

Значение *flags*	Смысл
T_EXPEDITED	Отмечает сообщение как срочное. Действует аналогично флагу MSG_OOB в гнездах. Провайдер транспорта может поддерживать эту опцию, а может и не поддерживать. В последнем случае функция возвращает код неудачного завершения и переменной *t_errno* присваивается значение TNOTSUPPORT
T_MORE	Сообщает процессу-получателю о том, что сообщение, которое будет передано при следующем вызове *t_snd*, является продолжением текущего

В случае успеха рассматриваемая функция возвращает число символов, помещенных в буфер *buf*, которые переданы нормально, а в случае неудачи -1. Если дескриптор *fd* задан как неблокирующий и сообщение, находящееся в *buf*, невозможно доставить получателю сразу же, функция немедленно завершает свою работу. В этом случае возвращается значение -1 и переменной *t_errno* присваивается значение TFLOW.

Функция *t_sndudata* используется для передачи дейтаграммных сообщений через конечную точку транспортировки без установления соединения. Структура *t_unitdata* объявляется следующим образом:

```
struct t_unitdata
{
    struct netbuf  addr;
    struct netbuf  opt;
    struct netbuf  udata;
};
```

Здесь структура *udata* содержит сообщение, подлежащее передаче в конечную точку транспортировки, адрес которой указан в *addr*. Структура *opt* содержит зависящие от провайдера транспорта опции, используемые при доставке данного сообщения.

Отметим, что в вызове *t_sndudata* нет аргумента *flags*, с помощью которого передаваемое сообщение определялось бы как срочное.

В случае успешного выполнения рассматриваемая функция возвращает 0, а в случае неудачи возвращает -1. Если дескриптор *fd* задан как неблокирующий и если сообщение, находящееся в *udata*, невозможно доставить получателю сразу же, функция завершает свою работу немедленно. В этом случае возвращается -1 и переменной *t_errno* присваивается значение TFLOW.

Далее срочное сообщение (MSG1) посылается в конечную точку транспортировки, с которой уже установлено соединение:

```
char* MSG1 = "Hello World";
if (t_snd(MSG1, strlen(MSG1)+1, T_EXPEDITED) < 0) t_error("t_snd");
```

А в этом примере дейтаграммное сообщение (MSG1) посылается в конечную точку транспортировки с адресом 3 (локальное соединение):

```
char* MSG1 = "Hello World";
struct t_unitdata *t_ud =
        (struct t_unitdata*)t_alloc(fd, T_UNITDATA, T_ALL);
if (!t_ud) {
    t_error("t_alloc for T_UNITDATA fails");
    exit(1);
}
t_ud->addr.len = sizeof(int);      /* установить адрес получателя */
*(int)t_ud->addr.buf = 3;
t_ud->udata.len = strlen(MSG1)+1; /* указать сообщение,
предназначенное для передачи */
t_ud->udata.buf = MSG1;
if (t_sndudata(fd, t_ud) < 0) t_error("t_sndudata");
```

11.4.8. Функции t_rcv, t_rcvudata и t_rcvuderr

Прототипы функций *t_rcv*, *t_rcvudata* и *t_rcvuderr* выглядят следующим образом:

```
#include <tiuser.h>

int t_rcv ( int fd, char* buf, unsigned len, int* flags );
int t_rcvudata ( int fd, struct t_unitdata* udata, int* flags );
int t_rcvuderr ( int fd, struct t_uderr* uderr );
```

Функция *t_rcv* принимает сообщение, помещенное в буфер *buf* другим процессом, который соединен с транспортной конечной точкой, обозначенной аргументом *fd*. Эта точка должна быть виртуальным каналом и соединяться с другой конечной точкой посредством вызова *t_connect* (для клиентского процесса) или *t_accept* (для серверного процесса).

Значение *len* задает максимальный размер буфера *buf*. Аргумент *flags* — это адрес целочисленной переменной. Данная переменная содержит значение *flags*, которое посылается вместе с сообщением при вызове функции *t_snd*. Возможные значения, возвращаемые в *flags*,— 0, T_MORE и/или T_EXPEDITED (описаны в предыдущем разделе).

В случае успешного выполнения рассматриваемая функция возвращает число байтов данных, помещенных в *buf*, а в случае неудачи -1. Если точка *fd* задана как неблокирующая и никакое сообщение не может быть принято тотчас же, функция немедленно завершает свою работу, возвращает -1 и переменная *t_errno* устанавливается в TNODATA.

Функция *t_rcvudata* используется для приема дейтаграммных сообщений через конечную точку транспортировки, функционирующую в режиме без установления соединения. Значение *udata* — это адрес переменной типа *struct unitdata*, в которой содержится принятое дейтаграммное сообщение и адрес отправителя.

Аргумент *flags* содержит адрес целочисленной переменной, которой обычно присваивается значение 0. Если буфер *udata->udata.buf* слишком мал для приема всего сообщения, то значение *flags* устанавливается в T_MORE и ядро копирует ровно такую часть текста сообщения, какая может поместиться в буфере *udata->udata.buf*. Для приема оставшейся части сообщения процесс должен вызвать функцию *t_rcvudata* еще раз.

В случае успешного выполнения эта функция возвращает 0, а в случае неудачи -1. Если точка *fd* задана как неблокирующая и никакое сообщение не может быть принято тотчас же, функция немедленно прекращает свою работу и возвращает -1; переменной *t_errno* присваивается значение TNODATA.

Функция *t_rcvuderr* используется для приема сообщений об ошибках, связанных с дейтаграммным сообщением. Ее следует вызывать только в том случае, если вызов *t_rcv* возвращает код неудачного завершения. Структура *t_uderr* объявляется следующим образом:

```
struct t_uderr
{
  struct netbuf  addr;
  struct netbuf  opt;
  long           error;
};
```

Здесь переменная *addr* содержит адрес пункта назначения сообщения об ошибке, *opt* — это зависящие от провайдера транспорта параметры, используемые при доставке данного сообщения, а *error* — код ошибки.

Значение *uderr* может быть задано как NULL. Это означает, что диагностические сообщения об ошибках не нужны и функция просто сбрасывает внутренний флаг, сигнализирующий о наличии ошибки.

В случае успешного выполнения эта функция возвращает 0, а в случае неудачи возвращает -1.

Ниже принимается сообщение из конечной точки транспортировки, с которой установлено соединение:

```
int  flags;
char buf[80];
if (t_rcv(fd, buf, sizeof(buf), &flags) < 0) t_error("t_rcv");
```

А в этом примере принимается дейтаграммное сообщение из конечной точки транспортировки, предназначенной для однорангового соединения. Если вызов *t_rcvudata* неудачен, выводится диагностическое сообщение об ошибке:

```
struct t_unidata *t_ud =
                    (struct t_unidata*)t_alloc(fd, T_UNIDATA, T_ALL);
```

```
if (!t_ud) {
    t_error("t_alloc for T_UNIDATA fails");
    exit(1);
}
int   flags;
char buf[80];
t_ud->udata.len = sizeof(buf);      /* организовать буфер для приема
                                       сообщения */
t_ud->udata.buf = buf;

if (t_rcvudata(fd, t_ud, &flags) < 0)  {    /* принять дейтаграммное
                                               сообщение */
   if (t_errno==TLOOK)  {
      struct t_uderr *uderr =
            (struct t_uderr*)t_alloc(fd, T_UDERROR, T_ALL);
      if (!uderr)   {
         t_error("t_alloc for T_UDERROR fails");
         exit(2);
      }
      if (t_rcvuderr(fd, uderr) < 0)  /* получить код ошибки */
         t_error("t_uderr");
      else cerr << "Error code is:" << uderr->error <<endl;
      t_free(uderr, T_UDERROR);     /* удалить запись с данными
                                       об ошибках */
   }
   else t_error("t_rcvudata");
} else cout << "receive msg:" << buf << "\n";
```

11.4.9. Функции t_sndrel, t_rcvrel

Прототипы функций *t_sndrel* и *t_rcvrel* выглядят следующим образом:

```
#include <tiuser.h>

int t_sndrel ( int fd );
int t_rcvrel ( int fd );
```

Функция *t_sndrel* передает запрос в используемый провайдер транспорта на разрыв соединения только в одном направлении. Процесс не может посылать сообщения в конечную точку транспортировки, обозначенную аргументом *fd*, но может продолжать принимать сообщения через эту точку (до получения подтверждения о приеме запроса на разрыв соединения).

В случае успешного выполнения эта функция возвращает 0, а в случае неудачи возвращает -1. Если точка *fd* задана как неблокирующая и запрос на разрыв соединения в одном направлении не может быть передан в используемый провайдер транспорта немедленно, функция возвращает -1 и переменной *t_errno* присваивается значение TFLOW.

Функция *t_rcvrel* подтверждает прием запроса на разрыв соединения. После вызова этой функции процесс не должен пытаться принимать сообщения через конечную точку транспортировки, указанную аргументом *fd*. В случае успешного выполнения данная функция возвращает 0, а в случае неудачи возвращает -1.

Функции *t_sndrel* и *t_rcvrel* поддерживаются не всеми провайдерами транспорта. Если провайдер не поддерживает эти функции, а они вызываются, то возвращается значение -1 и переменной *t_errno* присваивается значение TNOTSUPPORT.

В следующем примере показано, как в конечную точку транспортировки посылается запрос на разрыв соединения в одном направлении. Затем программа ждет подтверждения уведомления о разрыве соединения:

```
if (sndrel(fd) <0)
   t_error("sndrel");
else if (t_rcvrel(fd) < 0) t_error("rcvrel");
```

11.4.10. Функции t_snddis, t_rcvdis

Прототипы функций *t_snddis* и *t_rcvdis*:

```
#include <tiuser.h>

int t_snddis ( int fd, struct t_call* call );
int t_rcvdis ( int fd, struct t_discon* conn );
```

Функция *t_snddis* служит для аварийного разрыва установленного ранее транспортного соединения и отклонения клиентского запроса на соединение. При отклонении запроса на соединение в поле *call->sequence* указывается, какой запрос отклоняется. При аварийном разрыве соединения аргумент *call* может иметь значение NULL; в противном случае используется только поле *call->udata*, в котором содержатся пользовательские данные, передаваемые в удаленную транспортную точку вместе с уведомлением об аварийном разъединении.

В случае успешного выполнения эта функция возвращает 0, а в случае неудачи возвращает -1.

Функция *t_rcvdis* служит для отправки уведомления об аварийном разрыве соединения и отправки вместе с уведомлением пользовательских данных. Аргумент *conn* может иметь значение NULL, если процесс не запрашивает о пользовательских данных и причине преждевременного расторжения. В противном случае *conn* содержит адрес переменной типа *struct t_discon*. Эта структура объявляется следующим образом:

```
struct t_discon
{
  struct netbuf  udata;
  int            reason;
  int            sequence;
};
```

Поле *conn->udata* содержит пользовательские данные, переданные одновременно с вызовом *t_snddis*. Поле *conn->reason* содержит код причины разъединения, который зависит от провайдера транспорта. Поле *conn->sequence* имеет значение только для серверного процесса, выполнившего несколько вызовов *t_listen*, и используется для того, чтобы определить, какой клиентский процесс инициировал вызов *t_connect* и, затем, вызов *t_snddis*.

В случае успешного выполнения эта функция возвращает 0, а в случае неудачи возвращает -1.

В следующем примере в конечную точку транспортировки посылается запрос на аварийный разрыв соединения:

```
if (t_snddis(fd) < 0) t_error("t_snddis");
```

Ниже функция *t_rcvdis* используется для получения кода причины отклонения запроса на соединение:

```
if (t_connect(fd, call, call) < && t_errno==TLOOK)
   if (t_look(fd)==T_DISCONNECT) {
      struct t_discon *conn = (struct t_discon*)t_alloc(fd, T_DIS, T_ALL);
      if (!conn)
         t_error("t_alloc for T_DIS fails");
      else if (t_rcvdis(fd, conn) < 0)
         t_error("t_rcvdis");
      else cout << "Disconnect reason code:" << conn->reason << endl:
   )
```

11.4.11. Функция t_close

Прототип функции *t_close* выглядит следующим образом:

```
#include <tiuser.h>

int t_close ( int fd );
```

Функция *t_close* заставляет провайдер транспорта освободить все системные ресурсы, которые выделены для конечной точки транспортировки, указанной аргументом *fd*, и закрывает файл устройства, связанный с этим провайдером. Данную функцию следует вызывать после завершения соединения с транспортной конечной точкой, осуществленного посредством вызова *t_sndrel* или *t_snddis*.

В случае успешного выполнения рассматриваемая функция возвращает 0, а в случае неудачи возвращает -1.

Вот как можно закрыть конечную точку транспортировки:

```
if (t_close(fd) < 0) t_error("t_close");
```


11.5. Класс TLI

В этом разделе описан класс TLI, который выполняет функции, похожие на функции класса *sock* (см. раздел 11.2). В частности, класс TLI инкапсулирует все низкоуровневые интерфейсы системных вызовов TLI и обеспечивает управление динамической памятью, используемой для хранения TLI-данных (таких как *struct t_call*, *struct t_bind* и т.д.). Это позволяет сократить время обучения и программирования для тех пользователей, которые хотят с помощью интерфейса транспортного уровня решать задачи межпроцессного взаимодействия. Кроме того, класс TLI способствует максимальной унификации пользовательских приложений.

Класс TLI определяется в заголовке *tli.h* следующим образом:

```
#ifndef TLI_H
#define TLI_H

/* Определение класса TLI */
#include <iostream.h>
#include <unistd.h>
#include <string.h>
#include <tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <netdir.h>
#include <netconfig.h>

#define UDP_TRANS "/dev/ticlts"
#define TCP_TRANS "/dev/ticotsord"
#define DISCONNECT  -1

class tli {
    private:
      int     tid;          // дескриптор транспорта
      int     local_addr;   /* адрес транспорта для локального IPC */
      struct nd_addrlist  *addr; /* адрес транспорта для сетевого IPC */
      struct netconfig    *nconf;  /* файл устройства провайдера
                                      транспорта */
      int             rc;       // TLI functions return status code

        /* выделить структуру для передачи дейтаграммного сообщения
           в конечную точку сети Internet */
        struct t_unitdata* alloc_ud ( int nsid, char* service,
                                      char* host )
        {
              struct t_unitdata* ud=(struct t_unitdata*)t_alloc(
                      nsid==-1 ? tid : nsid, (T_UNITDATA), (T_ALL));
              if (!ud)
```

```
        {
            t_error("t_alloc of t_unitdata"); return 0;
        }
        struct nd_hostserv  hostserv;
        struct netconfig    *ncf;     // TI address
        struct nd_addrlist  *Addr;
        void                *hp;

        if ((hp=setnetpath()) == 0)
        {
           perror("Can't init network"); return 0;
        }
        hostserv.h_host = host;
        hostserv.h_serv = service;
        while ((ncf=getnetpath(hp)) != 0)
        {
            if (ncf->nc_semantics == NC_TPI_CLTS
                && netdir_getbyname(ncf, &hostserv, &Addr)==0)
            break;
        }
        endnetpath(hp);
        if (!ncf) {
            cerr << "Can't find transport for '" << service
                                                << "'\n";

            return 0;
        }
        ud->addr = *(Addr->n_addrs);
        return ud;
    };
    /* выделить структуру для передачи дейтаграммного сообщения
       в конечную точку */
    struct t_unitdata* alloc_ud ( int nsid, int port_no )
    {
        struct t_unitdata* ud=(struct t_unitdata*)t_alloc(
                nsid==-1 ? tid : nsid, (T_UNITDATA), (T_ALL));
        if (!ud)
        {
            t_error("t_alloc of t_unitdata"); return 0;
        }
        ud->addr.len = sizeof(int);
        *(int*)ud->addr.buf = port_no;
        return ud;
    };
public:
    /* конструктор, обеспечивающий создание конечной
       транспортной точки для локального IPC */
    tli( int srv_addr, int connless = 0 )
    {
        local_addr = srv_addr;
        nconf = 0;
        addr  = 0;
```



```
        if ((tid=t_open(connless ? UDP_TRANS : TCP_TRANS,
                O_RDWR, 0)) < 0)
            t_error("t_open fails");
    };
    /* конструктор, обеспечивающий создание конечной
       транспортной точки для сетевого IPC */
    tli( char* hostname, char* service, int connless=0 )
    {
        struct nd_hostserv  hostserv;
        void  *hp;
        int type = connless ? NC_TPI_CLTS : NC_TPI_COTS_ORD;

        local_addr = 0;

        /* найти провайдер транспорта для указанных хоста
           и сервиса */
        if ((hp=setnetpath()) == 0)
        {
            perror("Can't init network");
            exit(1);
        }
        hostserv.h_host = hostname;
        hostserv.h_serv = service;
        while ((nconf=getnetpath(hp)) != 0)
        {
            if (nconf->nc_semantics == type
                && netdir_getbyname(nconf, &hostserv, &addr)==0)
                break;
        }
        endnetpath(hp);

        if (nconf == 0 )
           cerr << "No transport found for service: '"
                << service << "'\n";
        else if ((tid=t_open(nconf->nc_device, O_RDWR, 0)) < 0)
           t_error("t_open fails");
    };
    /* функция-деструктор */
    ~tli()  { shutdown(); close(tid);  };
    /* проверить статус успешного выполнения конструктора */
    int good()    { return tid >= 0;  };
    /* позволить провайдеру транспорта присвоить имя конечной
       точке */
    int Bind_anonymous( )
    {
        return t_bind(tid, 0, 0);
    };
    /* присвоить имя конечной точке */
    int Bind()
    {
        struct t_bind  *bind;
```

```
    if ((bind= (struct t_bind*)t_alloc(tid, T_BIND,
                T_ALL))==0)
    {
        t_error("t_alloc for t_bind");
        return -1;
    }
    bind->qlen = 1;  /* максимальное количество
                        запросов на соединение */
    if (nconf)  {            // Internet-адрес
        bind->addr = *(addr->n_addrs);
    } else {                 // локальный адрес
        bind->addr.len = sizeof(int);
        *(int*)bind->addr.buf = local_addr;
    }
    if ((rc = t_bind(tid, bind, bind)) < 0)
       t_error("t_bind");
    else      /* отобразить фактически назначенный адрес */
       cerr << "bind: " << (*(int*)bind->addr.buf) << endl;
    return rc;
};
/* ждать запроса на соединение от клиентской конечной
   точки транспортировки */
int listen ( struct t_call*& call )
{
     if (!call && (call = (struct t_call*)t_alloc(tid,
         T_CALL, T_ALL))==0)
     {
         t_error("t_alloc");
         return -1;
     }
     if ((rc=t_listen(tid,call)) < 0) t_error("t_listen");
     return rc;
}
/* принять запрос на соединение от клиентской конечной
   точки транспортировки */
int accept ( struct t_call * call )
{
     /* создать новую конечную точку для связи с клиентом */
     int resfd;
     if (nconf) // Internet IPC
         resfd = t_open( nconf->nc_device, O_RDWR, 0);
     else       // Local IPC, must be connection-based
         resfd = t_open(TCP_TRANS, O_RDWR, 0);
     if (resfd < 0)  {
         t_error("t_open for resfd");
         return -1;
     }
     /* присвоить новой конечной точке произвольное имя */
     if (t_bind(resfd,0,0) < 0)
     {
```



```
         t_error("t_bind for resfd");
         return -2;
     }
     /* соединить новую конечную точку с клиентом */
     if (t_accept(tid, resfd, call) < 0)
     {
         if (t_errno == TLOOK)
         {
             if (t_rcvdis(tid, 0) < 0)
             {
                 t_error("t_rcvdis");
                 return -4;
             }
             if (t_close(resfd) < 0)
             {
                 t_error("t_close");
                 return -5;
             }
             return DISCONNECT;
         }
         t_error("t_acept");
         return -6;
     }
     return resfd;
};
/* инициализировать запрос на соединение с серверной
   конечной точкой транспортировки */
int connect()
{
    struct t_call *call;
    if ((call = (struct t_call*)t_alloc(tid, T_CALL,
         T_ALL))==0)
    {
        t_error("t_alloc");
        return -1;
    }
    if (nconf)
        call->addr = *(addr->n_addrs);
    else {
        call->addr.len = sizeof(int);
        *(int*)call->addr.buf = local_addr;
    }
    cerr << "client: connect to addr="
         << (*(int*)call->addr.buf) << endl;

    if ((rc=t_connect(tid,call,0)) < 0)
    {
        t_error("client: t_connect");
        return -2;
    }
    return rc;
};
```

```
/* передать сообщение в удаленную конечную точку
   транспортировки, с которой установлено соединение */
int write( char* buf, int len, int nsid=-1 )
{
    if ((rc=t_snd(nsid==-1 ? tid : nsid, buf, len, 0)) < 0)
        t_error("t_snd");
    return rc;
};
/* прочитать сообщение, поступившее из удаленной точки
   транспортировки, с которой установлено соединение */
int read( char* buf, int len, int& flags, int nsid=-1 )
{
    if ((rc=t_rcv(nsid==-1 ? tid : nsid, buf, len, &flags)) < 0)
        t_error("t_snd");
    return rc;
    };
/* передать дейтаграммное сообщение в удаленную конечную
   точку */
   int writeto( char* buf, int len, int flag,
                char* service, char* host,  int nsid=-1 )
        (
    struct t_unitdata* ud = alloc_ud(nsid,service,host);
    ud->udata.len = len;
    ud->udata.buf = buf;
    if ((rc=t_sndudata(nsid==-1 ? tid : nsid, ud)) < 0)
       t_error("t_sndudata");
    return rc;
};

/* передать дейтаграммное сообщение в конечную точку,
   расположенную на этой же машине */
int writeto( char* buf, int len, int flag, int port_no,
             int nsid=-1 )
{
    struct t_unitdata* ud = alloc_ud(nsid,port_no);
    ud->udata.len = len;
    ud->udata.buf = buf;
    if ((rc=t_sndudata(nsid==-1 ? tid : nsid, ud)) < 0)
        t_error("t_sndudata");
    return rc;
};
/*  передать дейтаграммное сообщение в удаленную конечную
    точку, адрес которой определен в ud */
int writeto( char* buf, int len, int flag,
             struct t_unitdata* ud, int nsid=-1 )
{
    ud->udata.len = len;
    ud->udata.buf = buf;
    if ((rc=t_sndudata(nsid==-1 ? tid : nsid, ud)) < 0)
        t_error("t_sndudata");
```



```
        return rc;
    };
    /* диагностическое сообщение об ошибке приема */
    void report_uderr( int nsid )
    {
        if ((t_errno) == (TLOOK)) (
            struct t_uderr    *uderr;
            if ((uderr=(struct t_uderr*)t_alloc(nsid==-1 ?
                 tid : nsid, T_UDERROR, T_ALL))==0)
            (
                t_error("t_alloc of t_uderr"); return;
            )
            if ((rc=t_rcvuderr(nsid==-1 ? tid : nsid, uderr)) < 0)
                t_error("t_rcvuderr");
            else cerr << "bad datagram. error=" << uderr->error
                      << endl;
            t_free((char*)uderr,T_UDERROR);
        }
        else t_error("t_rcvudata");
    }
/* принять дейтаграммное сообщение из удаленной конечной точки
   транспортировки */
int readfrom( char* buf, int len, int& flags,
              struct t_unitdata*& ud, int nsid =-1)
{
    if (!ud && (ud=(struct t_unitdata*)t_alloc(nsid==-1 ?
        tid : nsid, T_UNITDATA, T_ALL))==0)
    {
        t_error("t_alloc of t_unitdata"); return -1;
    }
    ud->udata.len = len;
    ud->udata.buf = buf;
    if ((rc=t_rcvudata(nsid==-1 ? tid : nsid, ud, &flags)) < 0)
        report_uderr(nsid);
    return rc;
};
/* закрыть соединение путем отправки уведомления
   о преждевременном разъединении */
int shutdown( int nsid = -1 )
    {
        return t_snddis( nsid==-1 ? tid : nsid, 0 );
    };
};      /* class TLI */
#endif
```

Открытые функции-члены класса TLI почти однозначно соответствуют открытым функциям-членам класса *sock*. Это обусловлено однозначным соответствием между API TLI и API гнезд.

Основное различие между классом TLI и классом *sock* состоит в правилах именования, установленных для каждого типа объекта. В частности, имя объекта *sock* может быть путевым UNIX-именем (для гнезда домена UNIX)

или комбинацией хост-имени и имени порта (для гнезда домена Internet). Имя объекта TLI может быть целым числом (для локального соединения) или комбинацией хост-имени и имени сервиса (для соединения, осуществляемого через Internet).

Еще одно различие между классом TLI и классом *sock* касается функции *listen*. В то время как функция *sock::listen* лишь устанавливает максимально допустимое для объекта *sock* количество запросов на соединение, функция *TLI::listen* используется для перехода в режим ожидания поступления клиентского запроса на соединение. Функция *sock::accept*, по сути дела, объединяет функции *TLI::listen* и *TLI::accept*.

Класс TLI не поддерживает неблокирующий режим, но пользователи могут, модифицировав его, получить такую поддержку. В следующих двух разделах описаны приложения IPC, в которых используется класс TLI.

11.6. Пример передачи сообщений по схеме клиент/сервер

Первый пример использования класса TLI — это новая версия приложения, представленного в разделе 10.3.7 (передача сообщений между клиентами и сервером). В этой версии используются конечные точки транспортировки с установлением соединения, с помощью которых формируется канал связи между сервером сообщений и клиентскими процессами. Поскольку сервер устанавливает связь непосредственно с каждым клиентским процессом, любое посылаемое от клиента сообщение должно представлять собой сервисную команду (например, LOCAL_TIME, UTC_TIME, QUIT_CMD и т.д.), введенную в виде строки символов. Сервер посылает свой ответ клиенту также в виде строки символов.

Ниже приведена программа сервера сообщений, имя которой *tli_msg_srv.C*.

```
#include "tli.h"
#include <sys/times.h>
#include <sys/types.h>

#define MSG1 "Invalid cmd to message server"

typedef enum { LOCAL_TIME, GMT_TIME, QUIT_CMD, ILLEGAL_CMD } CMDS;

/* обработать команды клиента */
void process_cmd (tli* sp, int fd )
{
    char    buf[80];
    time_t  tim;
    char*   cptr;
    int     flags;

    switch (fork()) {
```



```
        case -1:  perror("fork");  return;
        case 0:   break;
        default:  return;  // родительский процесс
    }

    /* читать команды клиента до поступления символа EOF
       или QUIT_CMD */
    while (sp->read(buf, sizeof buf, flags, fd) > 0)
    {
        cerr << "server: read cmd: '" << buf << "'\n";
        int     cmd = ILLEGAL_CMD;
        (void)sscanf(buf,"%d",&cmd);
        switch (cmd) {
            case LOCAL_TIME:
                tim = time(0);
                cptr = ctime(&tim);
                sp->write(cptr, strlen(cptr)+1, fd);
                break;
            case GMT_TIME:
                tim = time(0);
                cptr = asctime(gmtime(&tim));
                sp->write(cptr, strlen(cptr)+1, fd);
                break;
            case QUIT_CMD:
                sp->shutdown(fd);
                exit(0);
            default:
                sp->write(MSG1, sizeof MSG1, fd);
        }
    }
    exit(0);
}

int main( int argc, char* argv[])
{
    char  buf[80], socknm[80];
    int  port=-1, nsid, rc;
    fd_set  select_set;
    struct timeval timeRec;

    if (argc < 2) {
        cerr << "usage: " << argv[0] << " <service|no> [<host>_]\n";
        return 1;
    }

    /* проверить, указан ли номер порта */
    (void)sscanf(argv[1],"%d",&port);

    /* создать конечную точку с установлением соединения */
    tli  *sp;
    if (port==-1)
```

```
        sp = new tli( argv[2], argv[1] );
    else sp = new tli (port);


    if (!sp || !sp->good()) {
        cerr << "server: create transport endpoint object fails\n";
        return 1;
    }

    /* присвоить имя конечной точке транспортировки сервера */
    if (sp->Bind() < 0) {
        cerr << "server: bind fails\n";
        return 2;
    }

    for (struct t_call *call=0; sp->listen(call)==0; )
    {
        /* принять запрос на соединение, поступивший от клиента */
        if ((nsid = sp->accept(call)) < 0)
        {
            cerr << "server: accept fails\n";
            return 3;
        }

        t_free((char*)call, T_CALL); // освободить память

        cerr << "server: got one client connection. nsid=" << nsid <<"\n";
        /* создать порожденный процесс для обработки команд */
        process_cmd(sp,nsid);

        close(nsid); /* re-cycle file descriptor */
    }
    sp->shutdown();
    return 0;
}
```

Эта серверная программа вызывается с целочисленным адресом — для установления локального соединения или с именем сервиса и хоста — для установления Internet-соединения. Сервер начинает работу с создания конечной точки транспортировки и присвоения ей имени. Затем он входит в цикл и ожидает поступления запросов на соединение от клиентских процессов (вызывая для этого функцию *tli::listen*).

Для каждого полученного запроса на соединение сервер вызывает функцию *tli::accept*, которая устанавливает соединение с клиентской конечной точкой транспортировки. Эта функция возвращает также новый дескриптор (*nsid*) для того, чтобы дать возможность серверу взаимодействовать с клиентом, соединение с которым было установлено с ее помощью. Для обработки команд данного клиента сервер вызывает функцию *process_cmd*. Функция *process_cmd*, в свою очередь, создает для работы с клиентом порожденный

процесс. Сразу же после этого она возвращает управление серверному процессу, чтобы сервер мог продолжать мониторинг других запросов на соединение.

Каждый порожденный процесс, созданный функцией *process_cmd*, вызывает функцию *tli::read*, которая читает команды клиента, установившего соединение, и посылает ответы с помощью функции *tli::write*. Если клиент посылает в этот порожденный процесс команду QUIT_CMD, процесс разрывает соединение посредством вызова функции *tli::shutdown* и завершается.

Ниже приведена клиентская программа *tli_msg_cls.C*, обеспечивающая взаимодействие с программой-сервером путем использования объектов класса TLI:

```
#include "tli.h"
#define QUIT_CMD 2

int main( int argc, char* argv[])
{
    if (argc < 2) {
        cerr << "usage: " << argv[0] << " <service|no> [<host>]\n";
        return 1;
    }

    char buf[80];
    int  port=-1, rc, flags;

    /* проверить, указан ли целочисленный адрес или имя сервиса */
    (void)sscanf(argv[1],"%d",&port);

    tli  *sp;
    if (port==-1)
        sp = new tli( argv[2], argv[1] );
    else sp = new tli (port);

    if (!sp || !sp->good()) {
        cerr << "client: create transport endpoint object fails\n";
        return 1;
    }

    if (sp->Bind_anonymous() < 0) {
        cerr << "client: bind fails\n";
        return 2;
    }

    /* установить соединение с конечной точкой сервера */
    if (sp->connect() < 0) {
        cerr << "client: connect fails\n";
        return 3;
    }

    /* послать серверу команды 0 - 2 */
```

```
    for (int cmd=0; cmd < 3; cmd++)
    {
        /* сформировать команду для сервера */
        sprintf(buf,"%d",cmd);
        if (sp->write(buf,strlen(buf)+1) < 0) return 4;

        /* выйти из цикла, если QUIT_CMD */
        if (cmd==QUIT_CMD) break;

        /* прочитать ответ сервера */
        if (sp->read(buf,sizeof buf, flags) < 0) return 5;
        cerr << "client: recv '" << buf << "'\n";
    }
    sp->shutdown();
    return 0;
}
```

Эта программа-клиент вызывается с адресом сервера, значение которого может быть целым числом в случае локального соединения или именем сервиса и хоста при установлении Internet-соединения. Клиент начинает работу с создания конечной точки транспортировки и присвоения ей имени *anonymous*. Клиент не обязан присваивать точке конкретное имя, потому что ни один другой процесс не должен инициировать запрос на соединение по указанному адресу с серверным процессом.

После создания конечной точки транспортировки клиент вызывает функцию *tli::connect*, которая создает соединение типа "виртуальный канал" с серверной конечной точкой транспортировки. Если данная операция выполняется нормально, клиент посылает серверу ряд команд (посредством вызовов функции *tli::write*). Команды могут быть такими: LOCAL_TIME, UTC_TIME и QUIT_CMD. Передав очередную команду (это не относится к команде QUIT_CMD), клиент ожидает ответа сервера (для чего вызывает функцию *tli::read*) и направляет его ответное сообщение на стандартный вывод.

После передачи серверу команды QUIT_CMD клиент разрывает соединение посредством вызова функции *tli::shutdown*, а затем завершает работу.

Результат взаимодействия описанных программ представлен ниже:

```
%  CC -o tli_msg_srv tli_msg_srv.C -lnsl
%  CC -o tli_msg_cls tli_msg_cls.C -lnsl
%  tli_msg_srv 2 &
[1] 781
server: t_bind: 2
%  tli_msg_cls 2
client: connect to addr=2
server: got one client connection. nsid=4
server: read cmd: '0'
client: recv 'Fri Feb 17 22:34:50 1997'
server: read cmd '1'
client: recv 'Sat Feb 18 06:34:50 1997'
```



```
server: read cmd: '2'
[1] + Done   tli_msg_srv 2
```

Обе эти программы способны устанавливать соединение по сети Internet, причем никакая их доработка не требуется. Для того чтобы указанные программы можно было выполнять, достаточно ввести в файл */etc/services* новую запись:

```
test    4045/tcp
```

Посредством этой записи организуется сервис с именем *test*, который использует в качестве провайдера транспорта протокол TCP. Этому сервису назначен порт номер 4045. Ниже приведены результаты выполнения этих же программ, но с указанием адреса хост-имени *fruit* и имени сервиса *test*:

```
% hostname
fruit
%  tli_msg_srv test fruit &
[1] 776
server: t_bind: 135122
%  tli_msg_cls test fruit
client: connect to addr=135122
server: got one client connection. nsid=4
server: read cmd: '0'
client: recv 'Fri Feb 17 29:34:50 1997'
server: read cmd: '1'
client: recv 'Sat Feb 18 09:34:50 1997'
server: read cmd: '2'
[1] + Done   tli_msg_srv test fruit
```

11.7. Пример взаимодействия процессов с помощью дейтаграмм

Далее рассмотрим два одноранговых процесса, взаимодействующих через две конечные точки транспортировки, использующие для обмена сообщениями дейтаграммы. Эти программы — *tli_clts1* и *tli_clts2* — созданы соответственно из исходных файлов *tli_clts1.C* и *tli_clts2.C*.

Вот текст программы *tli_clts1.C*:

```
#include <sys/systeminfo.h>
#include "tli.h"

#define MSG1 "Hello MSG1 from clts1"

/* получить имя хоста */
int gethost( int argc, char* argv[], char host[], int len)
{
    if (argc!=3) {
        if (sysinfo(SI_HOSTNAME,host,len)< 0) {
```

```
            perror("sysinfo");
            return -1;
        }
    }
    else strcpy(host,argv[2]);
    return 0;
}

int main( int argc, char* argv[])
{
    char  buf[80], host(80);
    int  port=-1, rc, flags=0;

    if (argc < 2) {
        cerr << "usage: " << argv(0) << " <service|port_no>
                                        (<hostname>]\n";
        return 1;
    }

    /* проверить, указан ли номер порта */
    (void)sscanf(argv[1],"%d",&port);

    tli  *sp;
    if (port==-1)  {
        if (gethost(argc, argv, host, sizeof host) < 0) return 2;
        sp = new tli( host, argv[1], 1 );
    } else sp = new tli (port, 1);

    if (!sp || !sp->good()) {
        cerr << "cltsl: create transport endpoint object fails\n";
        return 3;
    }

    /* присвоить имя конечной точке транспортировки */
    if (sp->Bind() < 0) {
        cerr << "cltsl: bind fails\n";
        return 4;
    }

    struct t_unitdata *ud = 0;
    if (sp->readfrom( buf, sizeof buf, flags, ud) < 0)
    {
        cerr << "cltsl: readfrom fails\n";
        return 5;
    }
    cerr << "cltsl: read msg: '" << buf << "'\n";

    if (sp->writeto(MSG1, strlen(MSG1)+1, flags, ud) < 0)
    {
        cerr << "cltsl: writeto fails\n";
        return 6;
    }
```



```
    if (sp->readfrom(buf, sizeof buf, flags, ud) < 0)
    {
        cerr << "clts: readfrom fails\n";
        return 7;
    }
    cerr << "cltsl: read msg: '" << buf << "'\n";

    return 0;
}
```

Программа *tli_clts1* может вызываться с одним адресом, значение которого является целым числом (в случае установления локального соединения) или с именем сервиса и необязательным хост-именем (при использовании Internet-соединения). Если указывается имя сервиса, оно должно быть занесено в файл */etc/services*. Если хост-имя одновременно с именем сервиса не указывается, в качестве такового подразумевается хост-имя машины, на которой выполняется рассматриваемая программа.

Работа программы *tli_clts1* начинается с создания объекта TLI (конечной точки транспортировки) по заданным аргументам командной строки. Если необходимо создать конечную точку транспортировки для Internet, то вызывается функция *gethost*, которая выявляет хост-имя локальной машины в аргументе командной строки (если он есть) или посредством вызова функции *sysinfo*.

После создания конечной точки TLI и присвоения ей имени процесс читает сообщение (MSG2), посланное ему процессом *tli_clts2*. Получив сообщение MSG2, процесс направляет его на стандартный вывод. Затем он посылает в процесс *tli_clts2* сообщение MSG1, пользуясь адресом, содержащимся в переменной *ud*. Значение этой переменной присваивается посредством вызова функции *readfrom*.

Наконец, процесс ждет, когда *tli_clts2* пошлет последнее сообщение, MSG3. Получив данное сообщение, процесс направляет его на стандартный вывод и завершается. Конечная точка транспортировки, созданная процессом, уничтожается функцией-деструктором *TLI::~TLI*.

Ниже приведена программа *tli_clts2.C*, которая взаимодействует с программой *tli_clts1*:

```
#include <sys/systeminfo.h>
#include "tli.h"

#define MSG2 "Hello MSG2 from clts2"
#define MSG3 "Hello MSG3 from clts2"

typedef enum ( LOCAL_TIME, GMT_TIME, QUIT_CMD, ILLEGAL_CMD ) CMDS;

/* получить имя хоста */
int gethost( int argc, char* argv[], char host(], int len)
{
    if (argc!=4) {
        if (sysinfo(SI_HOSTNAME,host,len)< 0) {
```

```
            perror("sysinfo");
            return -1;
        }
    }
    else strcpy(host,argv[3]);
    return 0;
}

int main( int argc, char* argv[])
{
    char  buf(80], host[80];
    int  port=-1, cltsl_port=-1, rc, flags=0;

    if (argc < 2) {
        cerr << "usage: " << argv[0] << " <service|port_no>
                                <cltsl_service|no> [<host>]\n";
        return 1;
    }

    /* проверить, указан ли номер порта */
    (void)sscanf(argv[1],"%d",&port);
    (void)sscanf(argv[2],"%d",&cltsl_port);

    tli  *sp;
    if (port==-1)  {
        if (gethost(argc, argv, host, sizeof host) < 0) return 2;
        sp = new tli( host, argv(1], 1 );
    } else sp = new tli (port, 1);

    if (!sp || !sp->good()) {
        cerr << "clts2: create transport endpoint object fails\n";
        return 2;
    }

    /* присвоить имя конечной точке транспортировки */
    lf (sp->Bind() < 0) {
        cerr << "clts2: bind fails\n";
        return 3;
    }

    if (port==-1)
        rc = sp->writeto(MSG2,strlen(MSG2)+1, 0, argv[2], host);
    else rc = sp->writeto(MSG2, strlen(MSG2)+1, 0, cltsl_port);
    if (rc < 0)
    {
        cerr << "clts2: writeto fails\n";
        return 4;
    }

    struct t_unitdata *ud = 0;
    if (sp->readfrom(buf, sizeof buf, flags, ud) < 0)
    {
```



```
        cerr << "clts2: readfrom fails\n";
        return 5;
    }
    cerr << "clts2: read msg: '" << buf << "'\n";

    if (sp->writeto(MSG3, strlen(MSG3)+1, flags, ud) < 0)
    {
        cerr << "clts2: writeto fails\n";
        return 6;
    }
    return 0;
}
```

Программа *tli_clts2* очень похожа на программу *tli_clts1*. Отличие состоит в том, что она может вызываться не в одном, а в двух вариантах: во-первых, с назначенным целочисленным адресом и целочисленным адресом процесса *tli_clts1* (для случая локального соединения); во-вторых, с именем своего сервиса и именем сервиса *tli_clts1*, за которым может следовать необязательное хост-имя машины (при установлении Internet-соединения). Если указывается имя сервиса, это имя должно быть занесено в файл */etc/services*. Если хост-имя не указывается, берется хост-имя локальной машины.

Работа программы *tli_clts2* начинается с создания конечной точки TLI по заданным аргументам командной строки. Если необходимо создать конечную точку транспортировки для Internet, то вызывается функция *gethost*, которая выявляет хост-имя локальной машины из аргумента командной строки (если он есть) или посредством вызова функции *sysinfo*.

После создания конечной точки TLI процесс присваивает ей имя, а затем посылает сообщение MSG2 в процесс *tli_clts1*. После передачи сообщения MSG2 процесс ждет, когда процесс *tli_clts1* пошлет ему сообщение MSG1. По его получении процесс направляет сообщение на стандартный вывод.

Наконец, процесс посылает в процесс *tli_clts1* сообщение MSG1 и завершается. Конечная точка транспортировки, созданная процессом, уничтожается функцией-деструктором *TLI::~TLI*.

Приведенный ниже протокол отражает взаимодействие процессов *tli_clts1* и *tli_clts2*. Создаваемые в ходе взаимодействия конечные точки транспортировки имеют адреса, значения которых являются целыми числами. Конечной точке транспортировки процесса *tli_clts1* присваивается адрес 1, а процесса *tli_clts2* — адрес 2:

```
% CC -o tli_cltsl tll_cltsl.C -lns1
% CC -o tli_clts2 tli_clts2.C -lns1
% tli_cltsl 1 &
bind: 1
% tli_clts2 2 1
bind: 2
cltsl: read msg: 'Hello MSG2 from clts2'
clts2: read msg: 'Hello MSG1 from cltsl'
cltsl: read msg: 'Hello MSG3 from clts2'
[1] + Done    tli_cltsl 1
```

Чтобы эту программу можно было выполнять с использованием хост-имен и имен сервисов, в файле */etc/services* необходимо создать следующие записи:

```
utst1   4046/udp
utst2   4047/udp
```

Здесь *utst1* — это имя сервиса, используемого процессом *tli_clts1*, а *utst2* — имя сервиса, используемое процессом *tli_clts2*. Оба сервиса используют провайдер транспорта UDP, который предусматривает проведение сеанса связи без установления соединения.

Следующий протокол отражает взаимодействие этих же процессов, осуществляемое с использованием конечных точек транспортировки и сети Internet:

```
% tli_clts1 utst1 &
bind: 135123
% tli_clts2 utst2 utst1
bind: 135124
clts1: read msg: 'Hello MSG2 from clts2'
clts2: read msg: 'Hello MSG1 from clts1'
clts1: read msg: 'Hello MSG3 from clts2'
[1] + Done    tli_clts1 utst1
```

Обратите внимание: результат данного сеанса связи идентичен результату, полученному при работе с локальными конечными точками транспортировки. Перекомпилировать программы *tli_clts1.C* и *tli_clts2.C* не нужно. Эти же программы можно выполнять на разных машинах, включенных в локальную сеть. Предположим, что программа *tli_clts1* выполняется на машине с именем *fruit*, а программа *tli_clts2* — на машине *apple*. Как вызываются эти программы, описано ниже.

Запускаем программу *tli_clts1* на машине *fruit*:

```
fruit % tli_clts1 utst1 &
[1425]
```

Запускаем программу *tli_clts2* на машине *apple*:

```
apple % tli_clts2 utst2 utst1 fruit
```

После того как между компьютерами будет установлено соединение, выходные сообщения *tli_clts1* будут выводиться на экран машины *fruit*, а сообщения *tli_clts2* — на экран машины *apple*.

11.8. Заключение

В этой главе рассмотрены такие средства межпроцессного взаимодействия, как гнезда BSD UNIX и интерфейс транспортного уровня (TLI) UNIX System V.3/V.4. Гнезда и TLI более эффективны, чем сообщения, разделяемая память и семафоры, так как они позволяют взаимодействовать процессам,

работающим на разных машинах. Это очень важно для любого серьезного приложения архитектуры клиент/сервер, где сервер обычно работает на мощном компьютере, а клиентские процессы — на настольных машинах пользователей.

В данной главе подробно описывается синтаксис интерфейсов прикладного программирования гнезд и TLI. Приводятся примеры программ, иллюстрирующие применение этих механизмов. Определяются класс *sock* и класс *TLI*, инкапсулирующие эти API и позволяющие пользователям, которые решили применить такие конструкции для создания приложений IPC, сократить затраты времени на программирование.

Интерфейс транспортного уровня более гибок в применении, чем гнезда, потому что он поддерживает почти все транспортные протоколы. При организации взаимодействия с помощью гнезд может использоваться ограниченное число протоколов, причем гнезда разных типов поддерживают различные наборы протоколов. Кроме того, в TLI применяются более эффективные методы управления памятью (функции *t_alloc* и *t_free*), информирования об ошибках в пересылке сообщений (*t_error*, *t_rcvuderr*, *t_look*) и осуществления разрыва соединений (*tli_snddis*, *tli_rcvdis*, *tli_sndrel*, *tli_rcvrel*). Поэтому TLI позволяет пользователям создавать более развитые приложения для IPC.

Однако TLI используется только в UNIX System V.3 и V.4, тогда как гнезда представлены во всех последних разновидностях UNIX (BSD 4.2, 4.3, 4.4 и UNIX System V.4). Более того, на сегодняшний день уже имеется множество IPC-приложений, построенных на основе использования гнезд. Следовательно, если главными факторами для разработчиков приложений являются мобильность и совместимость с уже существующими приложениями, им следует ориентироваться на гнезда, а не на TLI.

ГЛАВА 12

Удаленные вызовы процедур

Удаленные вызовы процедур (Remote procedure calls, RPC) — это механизм, с помощью которого один процесс активизирует другой процесс на этой же или удаленной машине для выполнения какой-то функции от своего имени. RPC напоминает вызов локальной функции: процесс вызывает функцию и передает ей данные, а затем ожидает, когда она возвратит результат. Специфика заключается в том, что эту функцию выполняет другой процесс. Такое взаимодействие процессов обязательно протекает по схеме клиент/сервер, в которой процесс, активизирующий RPC, является клиентским, а процесс, выполняющий RPC-функцию,— серверным. Серверный процесс обеспечивает доступ к одной или нескольким сервисным функциям, которые могут вызываться его клиентами.

Удаленные вызовы процедур используются в сетевых приложениях для подключения к сетевым ресурсам других машин. Например, в распределенной базе данных серверным является процесс управления БД, обеспечивающий поиск и хранение данных в ее файлах. Клиентские процессы — внешние программы БД, которые позволяют пользователям запрашивать и обновлять данные. Клиентские процессы преобразуют команды пользователя в RPC и направляют их серверному процессу. Значения, возвращаемые RPC-функциями, сообщаются пользователю клиентскими процессами.

Рассмотрим еще один пример применения RPC: серверный процесс запускается на высокопроизводительном компьютере, а клиентские — на менее мощных. Когда клиентскому процессу нужно выполнить задание с большим объемом вычислений, он с помощью RPC активизирует серверный процесс, и задание выполняется на сервере. Таким образом обеспечивается более равномерная нагрузка на компьютеры и поддерживается приемлемый уровень производительности клиентского компьютера.

RPC отличаются и рядом других преимуществ:

- этот механизм "скрывает" от программистов основные детали передачи данных по сети, делая более легкими разработку и сопровождение программ, основанных на RPC;
- данные, которыми обмениваются клиентский и серверный процессы, представлены в формате, не зависящем от архитектуры компьютера, (таком как XDR), что позволяет взаимодействовать машинам с разными архитектурами (например, компьютерам с процессорами Intel x86 и рабочим станциям Sun SPARC);
- RPC поддерживают все сетевые транспортные протоколы (как с установлением соединений, так и без него);
- большинство развитых операционных систем (в частности, UNIX, VMS и Windows NT) поддерживают RPC и совместимы друг с другом, что позволяет разрабатывать сетевые приложения, которые могут выполняться на разных платформах под управлением разных операционных систем.

12.1. История создания RPC

В 80-е годы существовали разные методы реализации RPC, в частности Open Network Computing (ONC) фирмы Sun Microsystems и Network Computing Architecture (NCA) фирмы Apollo Computers. Сегодня в большинстве коммерческих разновидностей ОС UNIX, в том числе в HP-UX (Hewlett-Packard), AIX (IBM), Sun OS 4.1.x (Sun Microsystems), SCO UNIX (Santa Cruz Operations), RPC реализованы по методу ONC.

Однако в операционной системе Solaris 2.x (Sun Microsystems) и в ОС UNIX System V.4 для реализации RPC применяется модифицированная версия метода ONC. Оба метода очень похожи. Так, данные по сети передаются в формате внешнего представления данных (XDR), а для упрощения разработки RPC-приложений применяется компилятор *rpcgen*. Отличие состоит в том, что интерфейсы прикладного программирования RPC на базе ONC основаны на гнездах, тогда как API RPC UNIX System V.4 могут использовать как гнезда, так и TLI.

В этой главе рассматриваются приемы программирования RPC, поддерживаемые как методом ONC, так и методом UNIX System V.4. Большинство приведенных описаний относится к обоим методам.

12.1.1. Уровни интерфейса программирования RPC

Существуют разные уровни интерфейса программирования RPC. Наивысший уровень обеспечивает пользователям возможность вызывать системные функции RPC так же, как библиотечные функции C (например, *printf*), а низший позволяет создавать RPC-программы с помощью API RPC. Рассмотрим уровни интерфейса программирования RPC подробнее.

Интерфейс высшего уровня предоставляет пользователям возможность непосредственно вызывать системные функции RPC для сбора информации об удаленной системе. Эти функции можно использовать точно так же, как обычные библиотечные функции C, но при этом нужно указать специальные файлы заголовков, в которых объявляются прототипы этих функций, и связи между скомпилированными программами с помощью ключа *-lrpcsvc*. Объектный код этих библиотечных функций RPC содержится в библиотеке *librpcsvc.a*.

Преимущество библиотечных функций RPC состоит в том, что они легче в использовании и требуют меньше усилий при программировании. Однако в системе определено лишь несколько таких функций, поэтому область их применения ограничена.

Следующий уровень программирования RPC — это использование компилятора *rpcgen* для автоматического генерирования клиентских и серверных программ RPC. Пользователи должны написать только клиентские функции *main* (которые вызывают функции RPC) и серверные функции RPC. Компилятор *rpcgen* может генерировать и функции XDR, которые предназначены для преобразования используемых пользователем типов данных в формат XDR, необходимый для передачи информации между клиентом и сервером.

Преимущество использования компилятора *rpcgen* в том, что пользователи могут сконцентрировать внимание на написании функций RPC и клиентских главных функций. Знать низкоуровневые API RPC не нужно. Благодаря этому сокращаются затраты на программирование и снижается количество ошибок. Но этот подход имеет и недостатки: пользователи не могут управлять параметрами средств передачи данных, которые используются клиентскими и серверными программами, созданными компилятором *rpcgen*. Они не могут также управлять динамически распределяемой памятью, которая применяется функциями XDR.

Интерфейс низшего уровня обеспечивает создание клиентских и серверных программ RPC с помощью API RPC. Преимущество этого подхода заключается в том, что пользователи непосредственно управляют средствами передачи данных, которые применяются процессами, и динамически распределяемой памятью, используемой функциями XDR. Это, однако, требует дополнительных затрат, связанных с программированием.

12.2. Библиотечные функции RPC

Заголовок библиотечных функций RPC — <rpcsvc.h>. Каждый заголовок отвечает набору родственных библиотечных функций RPC и соответствующих функций XDR. Объектный код этих функций содержится в библиотеке *librpcsvc.a* в стандартном каталоге библиотек (например, */usr/lib*).

Ниже перечислены некоторые наиболее распространенные библиотечные функции RPC.

Библиотечная функция RPC	Назначение
rusers	Выдает количество пользователей, зарегистрированных в удаленной системе
rwall	Посылает сообщение удаленной системе
spray	Посылает удаленной системе заданное количество пакетов
rstat	Выдает данные о производительности удаленной системы

Рассмотрим библиотечные функции на примере функции *rstat*, которая определяет время работы одной или нескольких удаленных систем (т.е. время с момента начальной загрузки системы до текущего момента). Кроме того, функция *rstat* собирает статистические данные об удаленной системе. Эта функция взаимодействует с демоном *rc.rstatd*, работающим в удаленной системе, через механизм RPC:

```
/* rstat.C: выдать время работы удаленных систем */
#include <iostream.h>
#include <rpcsvc/rstat.h>

extern "C" enum clnt_stat rstat(char *host, struct statstime *statp);
int main( int argc, char* argv[] )
{
    struct statstime statv;
    if (argc==1) {
        cerr<<"usage:"<<argv[0]<<"<host>...\n";
        return 1;
    }
    while [--argc>0)  {
        if [rstat(*++argv,&statv)==RPC_SUCCESS) [
            int delta = statv.curtime.tv_sec - statv.boottime.tv_sec;
            int hour = delta/3600;
            int min  = delta%3600;
            cout<<"'"<<(*argv)<<"' up "<<hour<<"hr."
                <<[min/60)<<"min."<<[min%60)<<"sec."
                <<endl;
        }
        else perror("rstat");
    }
    return 0;
}
```

В качестве аргумента командной строки эта программа принимает одно или несколько хост-имен систем. Для каждой из указанных систем процесс вызывает функцию *rstat*, которая собирает статистические данные. Эти данные помещаются в переменную *statv*, а рабочее время вычисляется путем вычитания значения *statv.boottime.tv_sec* из значения *statv.curtime.tv_sec*. Тип данных *struct statstime* определяется в заголовке <rpcsvc/rstat.h>.

В результате выполнения программы может быть получена, например, следующая информация:

```
% CC rstat.C -o rstat -lrpcsvc -lnsl
% rstat fruit lemon
'fruit' up 1 hr. 12 min. 31 sec.
'lemon' up 0 hr. 39 min. 24 sec.
```

Как говорилось выше, пользоваться функциями RPC легко, а интерфейс программирования, который они предоставляют, подобен интерфейсу программирования библиотечных функций C. Однако число этих функций в системе ограничено, поэтому пользователям с помощью компилятора *rpcgen* или низшего уровня интерфейса программирования RPC приходится создавать для своих приложений дополнительные функции.

12.3. Компилятор rpcgen

Компилятор *rpcgen* имеется в большинстве UNIX-систем. Он обеспечивает разработку приложений на базе RPC. Входной информацией для компилятора является написанный пользователем текстовый файл, в котором содержится следующая информация:

- номер программы RPC;
- один или несколько номеров версий программы RPC;
- один или несколько номеров процедур RPC (в RPC термины *функция* и *процедура* эквивалентны);
- определенные пользователем типы данных, передаваемых с помощью RPC. Для каждого из этих типов данных *rpcgen* автоматически создает функции XDR;
- C-код (не обязательно), который должен копироваться прямо в выходные файлы, генерируемые компилятором.

Функция RPC обозначается номером программы, номером версии и номером процедуры.

RPC-программа соответствует одному серверному RPC-процессу, и этот процесс отвечает за выполнение от имени клиента всех необходимых процедур. Уровень обновления набора функций RPC задается номером RPC-версии — целым числом, которое должно начинаться с единицы. Номер RPC-процедуры — это уникальный идентификатор, присвоенный RPC-функции. Если существует несколько вариантов функции, то в обозначениях изменяется только номер версии, а номера программы и процедуры остаются без изменений. Номера процедур всех пользовательских функций RPC должны начинаться с единицы. Но всегда есть RPC-функция, номер процедуры которой в каждой RPC-программе равен нулю. Эта функция может автоматически генерироваться компилятором *rpcgen* или задаваться пользователем. Она не принимает никаких аргументов и ничего не возвращает. Эта функция проверяет, существует ли серверный процесс.

Рассмотрим, например, программу *print.c*:

```
/* print.C */
#include <iostream.h>
#include <fstream.h>

int print( char* msg )
{
    ofstream ofp("/dev/console" );
    if (ofp) {
          ofp <<msg<<endl;
          ofp.close();
          return 1;
     }
     return 0;
}

int main( int argc, char* argv[] )
{
    while (--argc > 0)
         if (print(*++argv))
            cout <<"msg`"<<(*argv)<<"`delivered OK\n";
         else cout<<"msg`"<<(*argv)<<"`delivered failed\n";
    return 0;
}
```

После компиляции и выполнения этой программы получим следующие результаты:

```
% CC print.C -o print
% print "Hello world" "Good-bye"
msg 'Hello world' delivered OK
msg 'Good-bye' delivered OK
```

Сообщения *Hello world* и *Good-bye* выводятся на системную консоль той машины, где выполняется программа *print*.

Чтобы преобразовать функцию *print* в удаленную процедуру, вручную создается файл *print.x*:

```
/* файл print.x: это входной файл для rpcgen */
program PRINTPROG
{
     version PRINTVER
     {
         int PRINT ( string ) = 1;
     } = 1;
} = 0x20000001;
```

В файле *print.x* функции *print* присвоены номер программы, номер версии и номер процедуры — 0x2000000001, 1 и 1 соответственно. *program* и *version* в *rpcgen* — зарезервированные ключевые слова, а PRINTPROG, PRINTVER и PRINT — определенные пользователем макросы для этих номеров в

контексте функции *print*. По соглашению эти константы обозначаются прописными буквами, но можно использовать и строчные.

Обратите внимание на объявление прототипа функции *print* в *print.x*: тип формального аргумента определен как *string*, который представляет собой определенный в RPC тип данных, соответствующий символьной строке, завершающейся NULL-символом. При помощи типа данных *string* механизм RPC различает данные типа символьных указателей от завершающихся нулем символьных массивов.

Компилятор *rpcgen* обрабатывает файл *print.x* следующим образом:

```
% ls
print.x
%  rpcgen print.x
%  ls
print.h   print.x   print_clnt.c   print_svc.c
```

Из файла *print.x* компилятор сгенерировал следующие три файла:

Файл	Назначение
print.h	Файл заголовка для клиентской и серверной программ
print_svc.c	Серверная программа без определения функции RPC
print_clnt.c	Модуль клиентской программы. Содержит все интерфейсные функции для вызова RPC-сервера

Если входной файл для rpcgen называется *<имя>.x*, то соответствующие выходные файлы компилятора, как правило, получают имена *<имя>.h*, *<имя>_svc.c* и *<имя>_clnt.c*.

Заголовок *print.h* содержит объявление макросов PRINTPROG, PRINTVER и PRINT и прототип функции *print*. Файл *print.h* для приведенного выше файла *print.x* выглядит так:

```
#ifndef      _PRINT_H_RPCGEN
#define      _PRINT_H_RPCGEN
#include     <rpc/rpc.h>
#define      PRINTPROG          ((unsigned long)(0x20000001)
#define      PRINTVER           ((unsigned long)(1)
#define      PRINT              ((unsigned long)(1)
extern int* print_1(char**, CLIENT*);
#endif       /*!_PRINT_H_RPCGEN*/
```

Обратите внимание на объявление функции *print* в файле *print.h*: имя функции представляет собой исходное имя, за которым следует знак подчеркивания и номер процедуры. Таким образом, RPC-имя функции *print* версии 1 — *print_1*. Кроме того, возвращаемое значение функции *print_1* указано как *int**, а не как *int*. Это типично для RPC-функций: аргумент и возвращаемое значение каждой RPC-функции передаются по адресу, поэтому если локальная функция принимает аргумент типа *char**, то ее RPC-партнер принимает аргумент типа *char***. То же самое можно сказать и о возвращаемых

значениях: если локальная функция возвращает значение типа *int*, ее RPC-партнер возвращает значение типа *int**.

Функция *print_1* создается пользователем вручную из функции *print* и определяется в отдельном файле *print_1.c* следующим образом:

```
/* файл print_1.c: определение серверной функции print */
#include <stdio.h>
#include "print.h"

int* print_1( char** msg )
{
      static int result;
      FILE* ofp = fopen ("/dev/console", "w");
      if (ofp) {
            fprintf( ofp, "%s\n", *msg );
            fclose( ofp );
            result = 1;
      }
      else result = 0;
      return &result;
}
```

Определения *print* и *print_1* различаются главным образом тем, что значения аргументов и возвращаемые значения второй функции являются указателями. Поэтому переменная *result* объявляется в *print_1* как статическая, чтобы эта функция могла возвратить ее адрес.

После определения функции *print_1.c* можно с помощью С-компилятора откомпилировать и запустить серверную программу:

```
% cc -o print_server print_1.c print_svc.c -lnsl
% print_server
```

Символ амперсанда ("&") при запуске серверной программы указывать не нужно, так как в *print_svc.c* определено, что эта программа должна выполняться в фоновом режиме автоматически. Опция *-lnsl* говорит о том, что для разрешения всех ссылок серверной программы на внешние библиотечные функции RPC нужно при компоновке загрузить библиотеку *libnsl.so* или *libnsl.a*. Эта опция характерна для ОС Solaris, и на других платформах для этой цели может использоваться другая опция.

Главная функция клиентской программы должна быть определена пользователем. Эта функция вызывает удаленную функцию *print_1*. Приведенная ниже функция *print_main.c* получена путем модификации функции *main* в *print.c*, в результате чего построена главная клиентская программа:

```
/* print_main.c: главная клиентская функция */
#include <stdio.h>
#include "print.h"
int main(int argc, char* argv[])
{
   int    *res, i;
   CLIENT *cl;
   if (argc<3) {
     fprintf( stderr, "usage: %s <svc_host> msg...\n", argv[0] );
     return 1;
   }
   if (!(cl = clnt_create( argv[1], PRINTPROG, PRINTVER, "tcp"))) {
     clnt_pcreateerror(argv[1]);
     return 2;
   }
   for (i=argc-1; i > 1; i--) {
        if (!(res = print_1(&argv[i], cl))) {
             clnt_perror(cl, argv[1]);
             return 3;
        }
        else if (*res==0) {
             fprintf( stderr, "print_1 fails\n" );
             return 4;
        }
        else printf( "print_1 succeeds for %s\n", argv[i] );
   }
   return 0;
}
```

Аргументами командной строки для клиентской программы являются хост-имя машины, на которой выполняется серверная программа *print*, и одно или несколько сообщений, которые надо передать серверу. С целью взаимодействия с этим сервером программа-клиент вызывает функцию *clnt_create*. Аргументами функции *clnt_create* являются хост-имя сервера, номер серверной программы и номер версии. Последний аргумент, *tcp*, указывает на то, что клиентский и серверный процессы будут взаимодействовать посредством транспортного протокола TCP/IP.

В случае неудачного завершения функция *clnt_create* возвращает NULL-указатель, после чего вызывается функция *clnt_pcreateerror*, которая выдает сообщение об ошибке. При нормальном выполнении функция *clnt_create* возвращает указатель CLIENT*, который используется в качестве значения второго аргумента при последующем вызове функции *print_1*. Функция *print_1* для клиентской программы определяется в файле *print_clnt.c*. Это фиктивная функция, которая, в свою очередь, вызывает функцию *print_1* серверной программы.

Если клиентская функция *print_1* возвращает NULL-указатель, это означает сбой вызова удаленной функции (по какой-то причине указанный механизм транспортировки не доступен или не работает). В этом случае вызывается функция *clnt_perror*, которая сообщает причину отказа. Если *print_1* возвращает не NULL-указатель, то проверяется указатель *res* для определения кода возврата функции *print_1*. Возможные значения кода состояния зависят от приложения, и для разных RPC-функций они разные.

Клиентская программа генерируется путем совместной компиляции модулей *print_main.c* и *print_clnt.c*:

```
% cc -o print_client print_clnt.c print_main.c -lnsl
```

Предположим, что программа *print_server* работает в фоновом режиме на машине *fruit*. Программу *print_client* можно запустить на любой машине, соединенной с машиной *fruit* локальной или глобальной сетью:

```
% print_client fruit "Hello world" "Good-bye"
print_1 succeeds for "Hello world"
print_1 succeeds for "Good-bye"
```

В окне системной консоли на машине *fruit* должны появиться такие сообщения:

```
Hello world
Good-bye
```

12.3.1. Функция clnt_create

Функция *clnt_create* имеет следующий синтаксис:

```
#include <rpc/rpc.h>

CLIENT*  clnt_create ( const char* hostname, const u_long prognum,
                       const u_long versnum, const char* nettype);
```

Переменная *hostname* — это символьная строка, заканчивающаяся NULL-символом. Она задает имя удаленной машины, на которой работает серверный процесс.

Переменные *prognum* и *versnum* — это соответственно номер программы и номер версии вызываемой удаленной функции.

Переменная *nettype* — это символьная строка, заканчивающаяся NULL-символом. Она указывает, какой механизм необходимо использовать для связи между клиентом и сервером. Ниже перечислены возможные значения этого аргумента.

Значение *nettype*	Смысл
"netpath"	Выбрать транспортный протокол из списка, заданного в переменной среды NETPATH. Если NETPATH не задана, выбрать транспортный протокол из условия nettype="visible" (т.е. из списка, который задан в файле */etc/netconfig*)
""	То же, что и "netpath"
"visible"	Выбрать транспортный протокол из тех элементов заданного в файле */etc/netconfig* списка, у которых установлен флаг "v"
"circuit_v"	То же, что и "visible", но выбрать транспортный протокол с установлением соединения
"datagram_v"	То же, что и "visible", но выбрать транспортный протокол без установления соединения
"circuit_n"	То же, что и "netpath", но выбрать транспортный протокол с установлением соединения
"datagram_n"	То же, что и "netpath", но выбрать транспортный протокол без установления соединения

Значение *nettype*	Смысл
"udp"	Использовать UDP
"tcp"	Использовать TCP

Если значение *nettype* равно "*netpath*", "", "*circuit_n*" или "*datagram_n*", то для определения транспортного протокола, который необходимо использовать для RPC-коммуникаций, клиентский и серверный процессы обращаются к переменной среды NETPATH. Переменная среды NETPATH определяется пользователем и содержит перечень разделенных двоеточиями названий транспортных протоколов. Ниже приведен пример команды shell, определяющей эту переменную:

```
% setenv NETPATH  tcp:udp
```

При неудачном завершении функция *clnt_create* возвращает NULL, а в случае успеха — указатель CLIENT*, который используется для связи с серверным процессом.

Функция *clnt_create* специфична для ОС UNIX System V.4. В ONC для создания указателей на RPC-клиентов применяются следующие функции:

```
#include <rpc/rpc.h>

CLIENT* clnttcp_create (struct sockaddr_in* svr_addr, const u_long prognum,
            const u_long versnum, int* sock_p, const u_long sendbuf_size,
            const u_long recvbuf_size);

CLIENT* clntudp_create (struct sockaddr_in* svr_addr, const u_long prognum,
            const u_long versnum, struct timeval retry_timeout, int* sock_p);
```

Функции *clnttcp_create* и *clntudp_create* — это версии *clnt_create* для TCP и UDP соответственно. Эти функции организуют взаимодействие клиентского и серверного процессов с использованием гнезд, и переменная NETPATH не задействуется. Значение аргумента *svr_addr* — указатель на адрес гнезда хост-имени сервера, а значение аргумента *sock_p* — указатель на номер порта гнезда RPC-сервера. Номер порта гнезда может быть задан как RPC_ANYSOCK, что означает порт, который фактически используется сервером.

Наконец, значение аргумента *retry_timeout* показывает, как долго клиентский процесс должен ожидать ответа сервера до повторной передачи запроса.

12.3.2. Программа rpcgen

Синтаксис вызова программы *rpcgen* выглядит следующим образом:

```
rpcgen [<опции>]<входной_файл>
```

Аргумент <*входной_файл*> — это путевое имя файла ***.x**, созданного пользователем. В этом файле указываются номер RPC-программы, номер версии (или номера версий) и номер процедуры (или номера процедур). Кроме того, в этом же файле определяется пользовательский тип данных, используемый в качестве входного аргумента и/или возвращаемого значения функций RPC.

При выполнении программы *rpcgen* можно задавать разные опции, среди которых самое важное значение имеют следующие.

Опция *rpcgen*	Назначение
-K <время>	Показывает, по истечении какого времени серверный процесс должен завершиться после обслуживания клиентского запроса. Если данная опция не задана, то используется ее значение по умолчанию, равное 120 секундам. Если установить значение опции <время> равным -1, то серверный процесс не завершится никогда
-s <механизм_транспортировки>	Задает механизм транспортировки, который должен использоваться для серверного процесса. Возможные значения опции <механизм_транспортировки> идентичны значениям *nettype* в вызове *clnt_create*

Рассмотрим, например, команду, которая вызывает *rpcgen* для компиляции файла *msg.x*. Эта серверная программа, созданная из соответствующего файла *msg_svc.c*, будет существовать в течение 60 секунд после обслуживания клиентского запроса. Для связи с клиентским процессом она будет использовать один из ориентированных на соединение транспортных протоколов категории *visible*, заданных в файле */etc/netconfig*:

```
% rpcgen -K 60 -s circuit_v msg.x
```

12.3.3. Получение списка файлов каталога с помощью программы rpcgen

В этом разделе мы рассмотрим еще один пример RPC-программы, генерируемой компилятором *rpcgen*. RPC-функция *scandir* в качестве входных аргументов получает путевое имя каталога и флаг, принимающий целочисленное значение. Она возвращает связный список имен файлов, которые существуют в указанном каталоге. Если значение входного флага не равно нулю, она также возвращает для каждого файла дополнительную информацию — UID владельца и время последней модификации.

Входной файл *scan.x* выглядит так:

```
/* scan.x: получение перечня файлов каталога */
const MAXNLEN = 255;

struct complex {
    unsigned ival;
    char     ary[80];
    int      *ptr;
    long     lval;
    float    dval;
};

typedef string name_t<MAXNLEN>;

typedef struct arg_rec *argPtr;

struct arg_rec {
    name_t   dir_name;
    int      lflag;
};

typedef struct dirinfo *infolist;

struct dirinfo {
    name_t    name;     /* имя каталога */
    u_int     uid;      /* UID          */
    u_long    modtime;  /* время последней модификации */
    infolist  next;     /* указатель на связный список */
};

union res switch (int errno) {
    case 0:  infolist  list;
    default: void;
};

program SCANPROG {
    version SCANVER {
        res SCANDIR(argPtr) = 1;
    } = 1;
} = 0x20000100;
```

Тип входного аргумента RPC-функции *scandir_1* — адрес указателя на переменную типа *struct arg_rec*, которая задает путевое имя каталога (поле *struct arg_rec::dir_name*) и целочисленный флаг (поле *struct arg_rec::lflag*). В случае успешного выполнения функции выводится связный список записей типа *struct dirinfo*, в каждой из которых содержится информация о файле. Определение *name_t* говорит о том, что имя файла — это символьная строка, содержащая максимум MAXNLEN символов и заканчивающаяся символом NULL.

Номер программы, номер версии и номер процедуры заданы как 0x20000100, 1 и 1.

Программа *rpcgen* компилирует *scan.x* при помощи следующей команды:

```
% rpcgen scan.x
```

В результате компиляции образуются четыре файла: *scan.h*, *scan_svc.c*, *scan_clnt.c*, *scan_xdr.c*. Последний файл содержит функции XDR-преобразования для значений данных типа *struct arg_rec* и *struct dirinfo*.

Функция *scandir_1* для этой серверной программы определяется в файле *scandir.c*:

```
#include <rpc/rpc.h>
#include <dirent.h>
#include <string.h>
#include <malloc.h>
#include <sys/stat.h>
#include "scan.h"

extern int errno;
res* scandir_l(argPtr* darg)
{
    DIR *dirp;
    struct dirent *d;
    infolist  nl, *nlp;
    struct stat statv;
    static res res;

    if (!(dirp = opendir((*darg)->dir_name))) {
        res.errno = errno; return &res;
    }
    xdr_free(xdr_res, (void*)&res);
    nlp = &res.res_u.list;
    while (d=readdir(dirp)) {
        nl = *nlp = (infolist)malloc(sizeof(struct dirinfo));
        nl->name = strdup(d->d_name);
        nlp = &nl->next;
        if ((*darg)->lflag) {
            char pathnm[256];
            sprintf(pathnm,"%s/%s",(*darg)->dir_name,d->d_name);
            if (!stat(pathnm,&statv)) {
                nl->uid = statv.st_uid;
                nl->modtime = statv.st_mtime;
            }
        }
    }
    *nlp = NULL;
    res.errno = 0;
    closedir(dirp);
    return &res;
}
```

Функция *scandir_1* вызывает API *opendir* для получения указателя на просматриваемый каталог, имя которого задано входным аргументом функции *(*darg)->dir_name*. В случае неудачи *opendir* функция *scandir_1* возвращает в поле *res.errno* значение *errno*.

Если вызов *opendir* завершается успешно, функция многократно вызывает *readdir* для получения имен всех файлов в указанном каталоге. Для каждого полученного файла динамически выделяется запись *struct dirinfo*. В эту запись функция заносит имя файла и, возможно, UID владельца и время последней модификации (если значение *(*darg)->lflag* не равно нулю). Записи *struct dirinfo* объединяются в связный список, указатель на который возвращается в поле *res.res_u.list*.

Серверная программа компилируется и запускается следующим образом:

```
% cc scan_l.c scan_xdr.c scan_svc.c -0 scan_svc -lnsl
% scan_svc
```

Клиентская главная программа находится в файле *scan_main.c*:

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

extern int errno;

int main( int argc, char* argv[])
{
    CLIENT *cl;
    char* server;
    struct arg_rec  *iarg = (struct arg_rec*)malloc(sizeof(struct arg_rec));
    res  *result;
    infolist nl;

    if (argc<3) {
        fprintf(stderr,"usage: %s host directory [<long>]\n", argv[0]);
        return 1;
    }
    server = argv[1];
    iarg->dir_name = argv[2];
    iarg->lflag = 0;
    if (argc==4 && sscanf(argv[3],"%u",&(iarg->lflag))!=1) {
        fprintf(stderr,"Invalid argument: '%s'\n", argv[3]);
        return 2;
    }

    if (!(cl = clnt_create(server, SCANPROG, SCANVER, "visible"))) {
        clnt_pcreateerror(server);
        return 3;
    }
    result = scandir_l(&iarg, cl);
    if (!result) {
        clnt_perror(cl, server);
        return 4;
    }
    if (result->errno) {
        errno = result->errno;
        perror(iarg->dir_name);
        return 5;
```

```
    }
    for (nl=result->res_u.list; nl; nl=nl->next) {
        if (iarg->lflag)
           printf("...%s, uid=%d, mtime=%s", nl->name, nl->uid,
                  ctime(&nl->modtime));
        else printf("...%s\n", nl->name);
    }
    return 0;
}
```

Клиентская программа вызывается с хост-именем серверного процесса, именем удаленного каталога и целочисленным флагом, который показывает, указывать ли для файлов заданного каталога UID владельца и время последней модификации (значение *lflag* не равно нулю) или этого делать не нужно (значение *lflag* равно нулю).

Клиентская функция *main* вызывает *clnt_create*, чтобы получить адрес конечной точки транспортировки для подключения указанного серверного процесса. После этого все данные входных аргументов упаковываются в динамически распределяемую память (на которую указывает переменная *iarg*), а затем вызывается RPC-функция *scandir_1*. По значению, возвращенному этой функцией, определяют, успешно ли выполнен данный RPC. Если вызов выполнен успешно, информация об удаленном файле направляется на стандартный вывод. В противном случае пользователь получает сообщение об ошибке.

Клиентская программа компилируется и запускается следующим образом (подразумевается, что сервер работает на машине *fruit*):

```
% cc scan_main.c scan_xdr.c scan_clnt.c -o scan_cls -lnsl
% scan_cls fruit  /etc  1
...magic, uid=2, mtime=Wed Aug 3 11:32:33 1997
...protocols, uid=10, mtime=Wed Aug 3 11:32:30 1997
```

12.3.4. Недостатки компилятора rpcgen

Тот факт, что *rpcgen* скрывает от пользователей низкоуровневые API RPC, имеет не только преимущества, но и недостатки.

К преимуществам *rpcgen* можно отнести сокращение затрат на программирование и уменьшение вероятности появления ошибок. Кроме того, пользователи могут уделить больше внимания созданию RPC-функций и клиентских функций *main*, а не функций транспортного интерфейса RPC.

В то же время компилятору *rpcgen* присущи следующие недостатки:

- пользователи не могут непосредственно управлять теми параметрами передачи данных, которые используют серверные и клиентские программы, генерируемые компилятором;
- пользователи не могут управлять динамической памятью, которой пользуются XDR-функции, генерируемые компилятором;

- большинство компиляторов *rpcgen* не генерируют C++-совместимые клиентские и серверные фиктивные функции. Для того чтобы сделать эти фиктивные функции приемлемыми для компилятора C++, может понадобиться ручная доводка (отметим, что *rpcgen* на рабочих станциях фирмы Sun Microsystems имеет опцию -C, которая позволяет генерировать C++-совместимые файлы).

Учитывая это, пользователям следует изучать низкоуровневые интерфейсы прикладного программирования RPC. Это позволит им преодолеть перечисленные выше ограничения, которые могут стать серьезным препятствием для разработки приложений.

12.4. Низкоуровневый интерфейс программирования RPC

Низкоуровневые интерфейсы прикладного программирования RPC объявляются в заголовке <rpc/rpc.h>. Эти API обеспечивают создание клиентских и серверных процессов с заданными пользователем параметрами транспортировки данных, регистрацию RPC-функций в демоне *rpcbind*, вызов удаленных RPC-функций из клиентских процессов. С помощью этих API клиентские процессы могут задавать методы аутентификации для установления защищенных соединений с серверными процессами.

Прежде чем представить читателю эти низкоуровневые API удаленных вызовов процедур, мы рассмотрим методы создания XDR-функций. Это объясняется тем, что некоторые низкоуровневые API RPC требуют, чтобы для аргументов и возвращаемых значений RPC-функций были заданы фактические данные и соответствующие XDR-функции. Кроме того, создавая собственные XDR-функции, пользователи получают возможность непосредственно управлять распределением и освобождением динамической памяти.

12.4.1. Функции XDR-преобразования

При каждой операции обмена данными между клиентским и серверным процессами XDR-функция вызывается дважды. Данные, которые клиент передает в RPC-функцию, сначала преобразуются в формат XDR и лишь затем пересылаются по сети. Этот процесс преобразования называется *сериализацией*. Перед тем как RPC-функция-адресат получит эти данные, такая же XDR-функция вызывается уже на стороне сервера и преобразует эти данные из формата XDR в формат, принятый на хост-машине. Этот процесс называется *десериализацией*. Имеются встроенные базовые функции XDR-преобразования для большинства основных типов данных RPC. Эти базовые функции способны выполнять и сериализацию, и десериализацию. Кроме того, пользовательские XDR-функции (которые, в свою очередь, вызывают базовые XDR-функции) автоматически наследуют эти возможности. Перечислим встроенные базовые XDR-функции.

Тип данных RPC	XDR-функция
int	xdr_int
long	xdr_long
short	xdr_short
char	xdr_char
u_int	xdr_u_int
u_long	xdr_u_long
u_short	xdr_u_short
u_char	xdr_u_char
float	xdr_float
double	xdr_double
enum	xdr_enum
bool	xdr_bool
string	xdr_string
union	xdr_union
opaque	xdr_opaque

Типы данных *u_int*, *u_long* и т.д. — это беззнаковые версии типов данных *int*, *long* и т.д. Данные типа *bool* преобразуются компилятором *rpcgen* в данные типа *bool_t*, а тип *bool_t* определяется в С как

```
typedef enum { TRUE=1, FALSE=0 }        bool_t;
```

Тип данных *opaque* соответствует последовательности произвольных байтов. Его можно использовать для объявления массивов фиксированного или переменного размера, например:

```
opaque      x[56];
opaque      vx<56>;
```

Компилятор *rpcgen* преобразует эти определения следующим образом:

```
char x[56];
struct
{
   u_int  xv_len;    /* фактическая длина массива xc_val */
   char   *xc_val;   /* динамический массив */
} xv;
```

Если пользователи определяют собственные типы данных, они могут генерировать XDR-функции для этих типов с помощью *rpcgen* или же писать собственные функции XDR-преобразования. Например, если пользователь определяет тип данных *struct complex* следующим образом:

```
struct complex
{
     unsigned     uval;
     char         ary[80];
     int          *ptr;
     long         lval;
     double       dval;
};
```

то XDR-функция для *struct complex* будет выглядеть так:

```
boot_t xdr_complex(XDR* xdrs, struct complex* objp)
{
     if (!xdr_u_int(xdrs, &objp->uval)) return FALSE;
     if (!xdr_vector(xdrs, objp->,80, sizeof(char),
        (xdrproc_t)xdr_char)
              return FALSE;
     if (!xdr_pointer(xdrs, &objp->ptr, sizeof(int), (xdrproc_t)xdr_int))
              return FALSE;
     if (!xdr_long(xdrs, &objp->lval)) return FALSE;
     if (!xdr_double(xdrs, &objp->dval)) return FALSE;
     return TRUE;
}
```

Все XDR-функции возвращают значения типа *bool_t*: TRUE, если функция выполнена успешно, FALSE в противном случае. XDR-функция для типа *complex* состоит из вызова базовой XDR-функции, которая преобразует все поля записи *complex*. Обратите внимание на тот факт, что массив фиксированного размера *complex::ary* преобразуется встроенной XDR-функцией *xdr_vector*, которая имеет следующие аргументы: указатель на буфер, в котором содержатся преобразованные данные; адрес массива фиксированного размера; число элементов массива; размер каждого элемента и XDR-функция, преобразующая каждый элемент массива.

Член *complex::ptr* преобразуется еще одной встроенной XDR-функцией, *xdr_pointer*, которая имеет следующие аргументы: указатель на буфер, содержащий преобразованные данные; адрес указателя; размер данных, на которые он указывает, и XDR-функция для этих данных.

12.4.2. Низкоуровневые API удаленных вызовов процедур

Существуют два набора интерфейсов прикладного программирования RPC для создания на базе RPC клиентских и серверных программ (по одному для каждого вида). Эти API и последовательность их вызова в клиентском и серверном процессах показаны на рис. 12.1.

Надписью *clnt_<create_RPC_handle>* обозначен набор API RPC. Каждый API RPC из этого набора создает интерфейс к RPC-клиенту, который можно использовать для связи с RPC-сервером для указанных номера RPC-программы и номера версии. Эти API перечислены ниже; они различаются по степени детализации при указании сетевого транспортного протокола, используемого для связи с RPC-сервером.

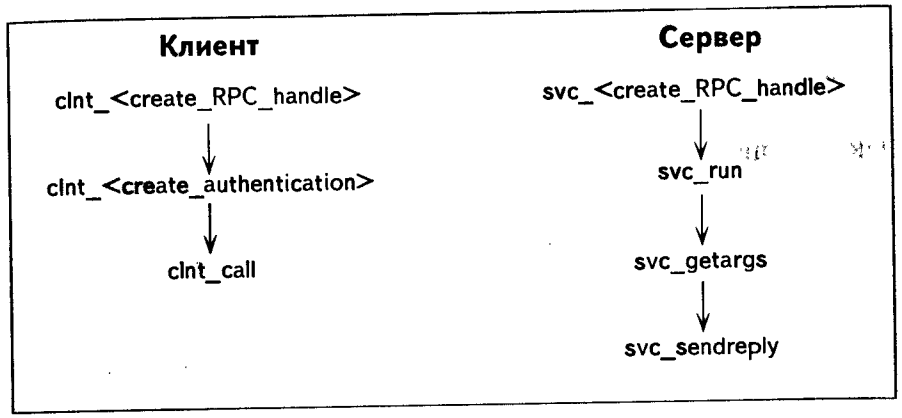


Рис. 12.1. Клиентские и серверные API удаленных вызовов процедур и последовательность их вызова

Клиентский API	Назначение
clnt_create	Задает базовый класс транспортных протоколов, которые будут использоваться во время выполнения
clnt_tp_create	Задает конкретный транспортный протокол
clnt_tli_create	Задает конечную точку транспортировки TLI, а также размеры передающего и принимающего буферов для RPC-связи. Описание TLI создается в клиентской программе

Надписью *clnt_<create_authentication>* обозначен набор API RPC, каждый из которых создает запись данных аутентификации, используемых RPC-сервером для аутентификации клиента. Эти API не обязательны и нужны только в том случае, если RPC-серверу требуется контроль доступа. Пользователи могут создавать собственные схемы аутентификации для RPC-связи между клиентом и сервером и собственные функции RPC-аутентификации для своих программ. Ниже перечислены стандартные функции RPC-аутентификации для клиентских программ.

Клиентский API	Назначение
authnone_create	Запись данных аутентификации содержит значение NULL. Вызывающий RPC-сервер не должен требовать аутентификации клиента
authsys_create_default	Запись данных аутентификации выполняется по методу управления доступом процессов, принятому в UNIX System V
authdes_seccreate	Запись данных аутентификации, зашифрованных по методу DES

Интерфейс *clnt_call* вызывает RPC-сервер для выполнения RPC-программы с заданным номером процедуры. Этот вызов включает входной аргумент

и его XDR-функцию, а также адрес переменной, которая принимает возвращаемое значение, и ее XDR-функцию.

На стороне сервера *svc_<create_RPC_handle>* обозначает набор API RPC, каждый из которых создает интерфейс к RPC-серверу, используемый для ответа на клиентские RPC-запросы. Эти API различаются по степени детализации при указании сетевого транспортного протокола, используемого для связи с RPC-клиентами. Перечислим их.

Серверный API	Назначение
svc_create	Задает базовый класс транспортных протоколов, которые выбираются во время выполнения
svc_tp_create	Задает конкретный транспортный протокол
svc_tli_create	Задает конечную точку транспортировки TLI, а также размеры передающего и принимающего буферов для RPC-связи. Описание TLI создается в серверной программе

API *svc_run* вызывается после того, как сервер создаст обработчик вызовов RPC. Эта функция входит в бесконечный цикл и ждет поступления клиентских RPC-запросов. Для обслуживания каждого вызова она активизирует указанную пользователем функцию-диспетчер. Эту функцию можно заменить пользовательской функцией, особенно если требуется, чтобы сервер, когда он не занят обслуживанием запросов, выполнял какие-нибудь другие операции.

Функция-диспетчер вызывает API *svc_getargs*, извлекающий аргументы RPC-функции, которая передана из клиентского процесса. Затем она вызывает затребованную RPC-функцию и с помощью API *svc_sendreply* передает возвращенное этой функцией значение клиенту.

Синтаксис этих API описан в следующих разделах. В разд. 12.5 рассматриваются два класса RPC, которые инкапсулируют низкоуровневые API удаленных вызовов процедур. Эти классы предоставляют пользователям упрощенный интерфейс программирования RPC, позволяя в то же время управлять механизмами RPC-транспортировки и памятью, используемой XDR-функциями для обработки данных.

12.5. Классы RPC

В данном разделе рассматриваются два класса RPC: один используется для построения RPC-сервера, а другой — для построения RPC-клиента. Эти классы позволяют разработчикам приложений работать с высокоуровневым RPC-интерфейсом, избавляя их от необходимости подробно изучать API удаленных вызовов процедур ONC и UNIX System V. Более того, пользователи могут создавать из этих RPC-классов собственные подклассы, объекты которых могут хранить больше данных, чем обработчики RPC-сервера и RPC-клиентов, и обеспечивают дополнительные функции (например, вызов предварительно определенных процедур при поступлении запросов и реализацию широковещательного режима).

Применение классов RPC создает дополнительные удобства и расширяет возможности пользователей:

- Классы RPC скрывают различия между API, построенными по методу ONC, и API, построенными по методу UNIX System V.4. Поэтому приложения, в которых используются эти классы, можно переносить на большинство коммерческих UNIX-систем.
- При создании RPC-приложений затрачивается меньше времени на обучение и программирование.
- Пользователи получают возможность сконцентрировать основные усилия на разработке базовых клиентских и серверных функций. Таким же достоинством, как вы помните, обладает компилятор *rpcgen*.
- Пользователи могут модифицировать классы RPC для управления используемыми механизмами RPC-транспортировки и методами аутентификации, а также для динамического распределения памяти, применяемой XDR-функциями.

В приведенном ниже заголовке *RPC.h* определены два класса RPC, которые инкапсулируют клиентские и серверные API RPC.

```
#ifndef _RPC_H
#define _RPC_H
/*-----------------------------------------------------------*/
/*  Определение заголовка для RPC-классов                    */
/*-----------------------------------------------------------*/

#include <iostream.h>
#ifdef __cplusplus
extern "C" {
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>

#ifdef SYSV4

#include <rpc/rpc.h>
#include <rpc/svc_soc.h>
#include <rpc/pmap_clnt.h>
#include <netconfig.h>

#else /* ONC */

#include <rpc/rpc.h>
#include <rpc/pmap_clnt.h>
#include <sys/socket.h>
```



```
#include <netdb.h>
#include <utmp.h>
#define AUTH_SYS    AUTH_UNIX
#define AUTH_SHORT  AUTH_UNIX
#endif

#ifdef __cplusplus
}
#endif

#define UNDEF_PROGNUM  0x0
#define TCP_BUFSIZ     4098

typedef int (*rpcprog)(SVCXPRT*);

typedef struct {
      unsigned  prgnum;
      unsigned  vernum;
      unsigned  prcnum;
      rpcprog   func;
} RPCPROG_INFO;

/*-----------------------------------------------------------*/
/* Класс RPC-сервера                                         */
/*-----------------------------------------------------------*/
class RPC_svc
{
            int              rc;
            unsigned         prgnum, vernum;
            static RPCPROG_INFO* progList;
            static int       numProg;
            SVCXPRT          *svcp;
      public:
            /* Программа-диспетчер */
            static void dispatch( struct svc_req* rqstp,
                  SVCXPRT  *xport )
            {
                  if (rqstp->rq_proc==NULLPROC) {
                        svc_sendreply(xport, (xdrproc_t)xdr_void, 0);
                        return ;
                  }
                  uid_t  uid = 0;
                  gid_t  gid = 0, gids[20];
                  short  len = 0;
                  switch (rqstp->rq_cred.oa_flavor)
                  {
                  case AUTH_NONE:
                        break;
                  case AUTH_SYS: {
#ifdef SYSV4
                  struct authsys_parms* authp;
                  authp = (struct authsys_parms*)rqstp->rq_clntcred;
```

```
#else
                struct authunix_parms* authp;
                authp = (struct authunix_parms*)rqstp->rq_clntcred;
#endif
                uid   = authp->aup_uid;
                gid   = authp->aup_gid;
            )   break;
#ifdef SYSV4
            case AUTH_DES:
            {
                if (!authdes_getucred(
                        (struct authdes_cred*)rqstp->rq_clntcred,
#ifdef SOLARIS_25
                          &uid, &gid, &len, (long*)gids))
#else
                          &uid, &gid, &len, (lnt*)gids))
#endif
                {
                    cerr << "decode DES auth. record failed\n";
                    svcerr_systemerr(xport);
                    return;
                }
            }   break;
#endif
            default:
                  svcerr_weakauth(xport);
                  return;
        }
        if (uid != getuid() && uid!=(uid_t)0)
        {
           cerr << "client (uid=" << uid << ") authentication fails\n";
           svcerr_weakauth(xport);
           return;
        }
        int i;
        for ( i=0; i < RPC_svc::numProg; i++)
           if (RPC_svc::progList(i].prcnum==rqstp->rq_proc &&
               RPC_svc::progList[i].vernum==rqstp->rq_vers &&
               RPC_svc::progList(i].prgnum==rqstp->rq_prog)
           {
              if ((*RPC_svc::progList[i].func)(xport)!=RPC_SUCCESS)
                  cerr << "rpc server execute prog " << rqstp->rq_proc
                      << " fails\n";
              break;
           }
        if (i >= RPC_svc::numProg) svcerr_noproc(xport);
    };
    /* Функция-конструктор. Создает объект RPC-сервера для
       заданного номера и версии программы */
    RPC_svc( unsigned prognum, unsigned versnum, const char* nettype)
    {
```



```
#ifdef SYSV4
        rc=svc_create(dispatch, prognum, versnum, nettype);
        if (!rc)
           cerr << "Can't create ROC server for prog: "
                << prognum << endl;
        else prgnum = prognum, vernum = versnum;
        svcp = 0;
#else
        int proto = 0;
        if (nettype && !strcmp(nettype,"tcp"))
        {
           svcp=svctcp_create(RPC_ANYSOCK, TCP_BUFSIZ, TCP_BUFSIZ);
           proto = IPPROTO_TCP;
        } else
        {
           svcp=svcudp_create(RPC_ANYSOCK);
           proto = IPPROTO_UDP;
        }

        if (!svcp)
        {
           rc = 0;
           cerr << "Can't create RPC server for prog: "
                << prognum << endl;
        } else
        {
           rc = 1;
           prgnum = prognum, vernum = versnum;
        }

        pmap_unset( prognum, versnum );
        if (!svc_register(svcp, prognum, versnum, dispatch, proto))
        {
           cerr << "could not register RPC program/ver: "
                <<  prognum << '/' <<  versnum << endl;
           rc = 0;
        }
#endif
    );
    /* Создать обработчик сервера для обратного вызова */
    RPC_svc( int fd, char* transport, u_long progno, u_long versno )
    {
#ifdef SYSV4

        struct netconfig *nconf = getnetconfigent(transport);
        if (!nconf)
        {
            cerr << "invalid transport: " << transport << endl;
            rc = 0;
            return;
        }
```

```
        svcp = svc_tli_create( fd, nconf, 0, 0, 0);
        if (!svcp)
        {
            cerr << "create server handle fails\n";
            rc = 0;
            return;
        }
        if (progno == UNDEF_PROGNUM)  /* generate a transient one */
        {
           progno = gen_progNum( versno, nconf, &svcp->xp_ltaddr);
        }
        if (svc_reg(svcp, progno, versno, dispatch, nconf)==FALSE)
        {
           cerr << "register prognum failed\n";
           rc = 0;
        }
         freenetconfigent( nconf );
#else
        /* fd должен содержать дескриптор гнезда, который может
           принимать значение RPC_ANYSOCK */
        if (progno == UNDEF_PROGNUM)  /* сгенерировать временный
                                         номер */
        {
            progno = gen_progNum ( versno, &fd, transport );
        }
        int      proto = 0;
        if (!strcmp(transport,"tcp"))
        {
            svcp = svctcp_create( fd, TCP_BUFSIZ, TCP_BUFSIZ );
            if (fd) proto = IPPROTO_TCP;
        }
        else
        {
            svcp = svcudp_create ( fd );
            if (fd) proto = IPPROTO_UDP;
        }

        if (!svcp)
        {
            cerr << "create server handle fails\n";
            rc = 0;
            return;
        }

        if (fd) pmap_unset( progno, versno );

        if (!svc_register(svcp, progno, versno, dispatch, proto))
        {
           cerr << "could not register RPC program/ver: "
                << progno << '/' << versno << endl;
           rc = 0;
        }
```



```
#endif
            prgnum = progno, vernum = versno;
            rc = 1;
        );

        /* Возвратить номер программы */
        u_long progno()    { return prgnum;     );

        /* Функция-деструктор */
        ~RPC_svc()
        {
             pmap_unset( prgnum, vernum );
             svc_unregister( prgnum, vernum );
             if (svcp) svc_destroy(svcp);
        );

        /* Проверка правильности создания объекта сервера */
        int good()         {   return rc;       };

        /* Сервер ожидает RPC-запросы от клиентов */
        static void run()  (   svc_run();       };

        /* Опрос на предмет наличия RPC-запросов. Это делается для
           асинхронного обратного вызова RPC  */
        static int  poll( time_t  tlmeout  )
        {
            int  read_fds = svc_fds;
            struct timeval  stry;
            stry.tv_sec = timeout;
            stry.tv_usec = 0;
#ifndef HPUX
            switch (select(32, (fd_set*)&read_fds, 0, 0, &stry))
#else
            switch (select(32, &read_fds, 0, 0, &stry))
#endif
            {
                case -1:   /*if (errno != EINTR) perror("poll");*/
                           return -1;
                case  0:   return 0;   /* нет событий */
                default:   svc_getreq( read_fds );
            }
            return 1;
        };

        /* Регистрация RPC-функции и начало обслуживания RPC-запроса */
        int run_func( int procnum, rpcprog func )
        {
           if (good())
           {
              if (func) add_proc( procnum, func );
              run();   /* функция ничего не возвращает */
           }
```

```
        return -1;
    };

    /* Регистрация новой RPC-функции */
    void add_proc( unsigned procnum, rpcprog func )
    {
        for (int i=0; i < numProg; i++}
            if (progList[i].func==func} return;
        if (++numProg == 1)
            progList = (RPCPROG_INFO*)malloc(sizeof(RPCPROG_INFO));
        else
            progList = (RPCPROG_INFO*)realloc((void*)progList,
                                sizeof(RPCPROG_INFO)*numProg);
        progList[numProg-1].func = func;
        progList[numProg-1].prgnum = prgnum;
        progList[numProg-1].vernum = vernum;
        progList[numProg-1].prcnum = procnum;
    };


    /* Вызывается RPC-функцией для получения значений аргументов
       от клиента */
    int getargs( SVCXPRT* transp, xdrproc_t func, caddr_t argp }
    {
        if (!svc_getargs( transp, func, argp}}
        {
            svcerr_decode(transp);
            return -1;
        } else return RPC_SUCCESS;
    };

    /* Вызывается RPC-функцией для отправки ответа клиенту */
    int reply( SVCXPRT* transp, xdrproc_t func, caddr_t argp )
    {
        if (!svc_sendreply(transp, func, argp}}
        {
            cerr << "Can't send reply\n";
#ifdef HPUX
        perror("svc_sendreply");
#else
            svcerr_systemerr(transp};
#endif
            return -1;
        } else return RPC_SUCCESS;
    };

#ifdef SYSV4
    /* Генерировать временный номер RPC-программы для асинхронного
       обратного вызова */
    static unsigned long gen_progNum( unsigned long versnum,
                        struct netconfig* nconf, struct netbuf*
```



```
addr)
    {
        static unsigned long transient_prognum = 0x5FFFFFFF;
        while (!rpcb_set( transient_prognum++, versnum, nconf,
addr)}
            continue;
        return transient_prognum -1;
    };
#endif
    static unsigned long gen_progNum ( unsigned long versnum,
                                       int* sockp, char* nettype )
    {
        static unsigned long transient_prognum = 0x5FFFFFFF;
        int  s, len, proto = IPPROTO_UDP, socktype = SOCK_DGRAM;
        struct sockaddr_in  addr;

        if (!strcmp(nettype,"tcp"})
        {
            socktype = SOCK_STREAM;
            proto    = IPPROTO_TCP;
        }

        if (*sockp== RPC_ANYSOCK)
        {
            if ((s = socket(AF_INET, socktype, 0)) < 0}
            {
                perror("socket");
                return 0;
            }
            *sockp = s;
        }
        else s = *sockp;

        addr.sin_addr.s_addr = 0;
        addr.sin_family      = AF_INET;
        addr.sin_port        = 0;
        len                  = sizeof(addr);

        (void)bind( s, (struct sockaddr*}&addr, len );
        if (getsockname( s, (struct sockaddr*)&addr, &len } < 0)
        {
            perror("getsockname");
            return 0;
        }
        while (!pmap_set( transient_prognum--, versnum, proto,
                addr.sin_port})
            continue;
        return transient_prognum -1;
    };
};
```

```
/*-----------------------------------------------------------*/
/* Класс RPC-объектов клиента                                */
/*-----------------------------------------------------------*/
class RPC_cls
{
      CLIENT  *clntp;
      char    *server;
    public:
      /* Функция-конструктор. Создает RPC-объект клиента для
         заданного сервера, номера программы и ее версии   */
      RPC_cls( char* hostname, unsigned prognum, unsigned vernum,
               char* nettype)
      {
#ifdef SYSV4
          if (!(clntp=clnt_create(hostname,prognum,vernum,nettype)))
               clnt_pcreateerror(hostname);
          else
          {
              server = new char[strlen(hostname)+1];
              strcpy(server,hostname);
          }
#else
          struct hostent* hp = gethostbyname(hostname);
          struct sockaddr_in  server_addr;
          int    addrlen, sock = RPC_ANYSOCK;

          if (!hp)
             cerr << "Invalid host name: '" << hostname << "'\n";
          else
          {
             addrlen = sizeof(struct sockaddr_in);
             bcopy( hp->h_addr, (caddr_t)&server_addr.sin_addr,
                                hp->h_length);
             server_addr.sin_family = AF_INET;
             server_addr.sin_port = 0;

             if (nettype && !strcmp(nettype,"tcp"))
                clntp=clnttcp_create(&server_addr, prognum, vernum,
                                &sock, TCP_BUFSIZ, TCP_BUFSIZ];
             else
             {
                struct timeval  stry;
                stry.tv_sec = 3;
                stry.tv_usec = 0;
                clntp=clntudp_create(&server_addr, prognum,
                                vernum, stry, &sock);
             }
```



```
             if (!clntp)
                clnt_pcreateerror(hostname);
             else
             {
                server = new char(strlen(hostname)+1];
                strcpy(server,hostname);
             }
          }
#endif
          if (clntp) set_auth ( AUTH_NONE );
      };

      /* destructor function */
      ~RPC_cls()  { (void)clnt_destroy( clntp ); };

      /* Проверка правильности создания клиентского объекта */
      int  good() ( return clntp ? 1 : 0; );

      /* Установить данные аутентификации */
      void set_auth( int choice, unsigned timeout = 60 )
      {
          switch (choice)
          {
             case AUTH_NONE:
                 clntp->cl_auth = authnone_create();
                 break;
             case AUTH_SYS:
             case AUTH_SHORT:
#ifdef SYSV4
                 clntp->cl_auth = authsys_create_default();
#else
                 clntp->cl_auth = authunix_create_default();
#endif
                 break;
#ifdef SYSV4
             case AUTH_DES:
             {
                 char netname(MAXNETNAMELEN+1];
                 des_block  ckey;
                 if (key_gendes(&ckey))
                 {
                     perror("key_gendes"];
                 }
                 if (!user2netname(netname, getuid(), "netcom.com"))
                 {
                    clnt_perror(clntp,server);
                 ]
                 else clntp->cl_auth =
                     authdes_seccreate(netname, timeout, server, &ckey);

                 if (!(clntp->cl_auth))
                 {
```

```
                    cerr << "client authentication setup fails\n";
                    perror("authdes_seccreate");
                    clnt_perror(clntp,server);
                    clntp->cl_auth = authnone_create();
                }
            }   break;
#endif
            default:
                cerr << "authentication method '" << (int)choice
                     << "' not yet supported\n";
                clntp->cl_auth = authnone_create();
        }
    };

    /* Вызов RPC-функции */
    int call( unsigned procnum, xdrproc_t xdr_ifunc, caddr_t argp,
              xdrproc_t xdr_ofunc, caddr_t rsltp,
              unsigned timeout = 20 )
    {
        if (!clntp) return -1;
        struct timeval timv;
        timv.tv_sec  = timeout;
        timv.tv_usec = 0;
        if (clnt_call(clntp, procnum, xdr_ifunc, argp,
                      xdr_ofunc, rsltp, timv)!=RPC_SUCCESS)
        {
            clnt_perror(clntp, server);
            return -2;
        }
        return RPC_SUCCESS;
    };

    /* Поддержка широковещательного режима */
    static int broadcast( unsigned prognum, unsigned versnum,
                          unsigned procnum, resultproc_t callback,
                          xdrproc_t xdr_ifunc, caddr_t argp,
                          xdrproc_t xdr_ofunc, caddr_t rsltp,
                          char* nettype = "datagram_v")
    {
#ifdef SYSV4
        return rpc_broadcast(prognum, versnum, procnum, xdr_ifunc,
                             argp, xdr_ofunc, rsltp, callback, nettype);
#else
        return clnt_broadcast(prognum, versnum, procnum, xdr_ifunc,
                              argp, xdr_ofunc, rsltp, callback);
#endif
    };

    /* Установить тайм-аут для клиента */
    int set_timeout( long usec )
    {
        if (!clntp) return -1;
```



```
        struct timeval timv;
        timv.tv_sec = 0;
        timv.tv_usec = usec;
        return clnt_control( clntp, CLSET_TIMEOUT, (char*)&timv);
    };

    /* Получить тайм-аут клиента */
    long get_timeout()
    {
        if (!clntp) return -1;
        struct timeval timv;
        if (clnt_control( clntp, CLGET_TIMEOUT, (char*)&timv)==-1)
        {
            perror("clnt_control");
            return -1;
        }
        return timv.tv_usec;
    };
};

#endif /* _RPC_H */
```

Последовательность вызова функций-членов этих классов для типичных клиента и сервера показана на рис. 12.2.

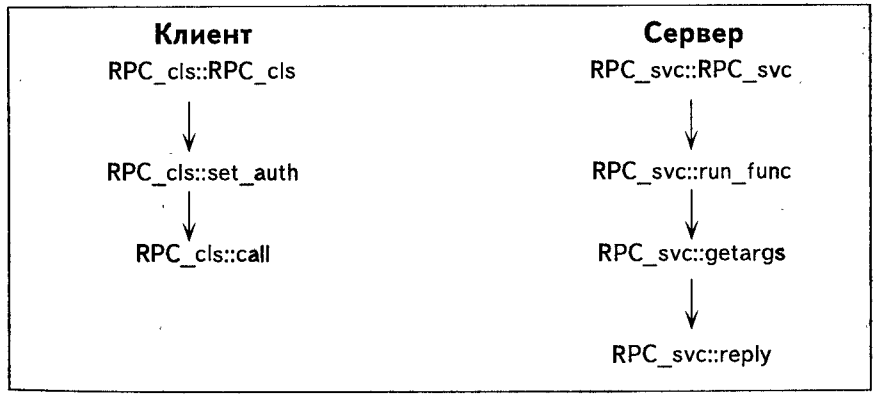


Рис. 12.2. Функции клиентского и серверного классов RPC и последовательность их вызова

Функция-конструктор *RPC_svc::RPC_svc* создает обработчик вызовов RPC-сервера для заданного номера RPC-программы и заданного номера RPC-версии. Кроме того, эта функция регистрирует пользовательскую функцию-диспетчер, которая активизируется, когда клиент вызывает RPC-функцию, управляемую сервером. Последний аргумент функции *RPC_svc::RPC_svc* — значение *nettype*. Оно задает транспортный протокол для связи между сервером и его клиентами. Функция *RPC_svc::RPC_svc* вызывает API *svc_create*, который создает интерфейс к серверному процессу. Пользователи могут

модифицировать функцию *RPC_svc::RPC_svc*, чтобы вместо *svc_create* она вызывала *svc_tp_create* или *svc_tli_create*.

Функция *RPC_svc::run_func* вызывается для регистрации на сервере RPC-функции и присвоенного ей номера процедуры, а затем блокирует сервер и переводит его в режим ожидания клиентских RPC-вызовов (через API *svc_run*). Если пользователь хочет зарегистрировать на сервере более одной RPC-функции, серверную программу необходимо соответствующим образом изменить:

```
RPC_svc svcp = new RPC_svc(...);
if (!svcp || svcp->good()) return 1;
svcp->add_func(<procnum1>, prog2);
.....
svcp->add_func(<procnumN-1>, progN-1);
svcp->run_func(<procnumN>, progN);
```

Когда поступает клиентский RPC-запрос, то вызывается зарегистрированная с помощью функции *RPC_svc::RPC_svc* функция-диспетчер, которая обслуживает клиентский запрос.

- Функция-диспетчер проверяет, равен ли затребованный клиентом RPC-номер нулю. Если равен, то клиент просто запрашивает сервер, а сервер просто посылает клиенту ответ с NULL-значением.
- Если клиент указывает какие-то аутентификационные данные, диспетчер проверяет их и при отрицательном результате выставляет флаг ошибки аутентификации. В данной функции аутентификация клиента выполняется по усмотрению сторон. Если речь идет о защите RPC-транзакций, то аутентификация обязательна. Сервер должен всегда требовать, чтобы клиенты посылали аутентификационные данные в каждом RPC-вызове. Метод RPC-аутентификации рассматривается в одном из следующих разделов.
- При положительном результате аутентификации диспетчер вызывает RPC-функцию, затребованную клиентом.

Каждая RPC-функция, вызываемая функцией *RPC_svc::dispatch*, должна иметь следующий прототип:

```
int <имя_функции> (SVCXPRT*);
```

где аргументом является указатель на обработчик для механизма транспортировки, используемого для связи с конечной точкой транспортировки клиента. В случае успешного выполнения функция возвращает нуль (RPC_SUCCESS), а в случае неудачи — ненулевое значение. Помимо этого данная RPC-функция должна обрабатывать глобальный указатель *RPC_svc** (имя его переменной определяется приложением), в котором находится адрес указателя на RPC-сервер. Функция сначала должна вызвать *RPC_svc::getargs*, чтобы извлечь аргументы из клиентского запроса, а затем *RPC_svc::reply*, чтобы передать клиенту возвращенное ею значение.

В клиентской программе обработчик клиента для заданных номеров программы и версии создается с помощью функции-конструктора *RPC_cls::RPC_cls*. Эта функция получает указатель на клиента путем внутреннего вызова API *clnt_create*. Как и функция-конструктор *RPC_svc*, функция *RPC_cls::RPC_cls* может быть модифицирована пользователями так, чтобы вызывать не *clnt_create*, а *clnt_tp_create* или *clnt_tli_create*.

Аутентификационные данные клиента можно установить, вызвав функцию *RPC_cls::set*. Класс *RPC_cls* поддерживает режимы AUTH_NONE, AUTH_SYS и AUTH_DES, но пользователи могут модифицировать функцию *RPC_cls::set_auth*, чтобы реализовать собственные методы аутентификации. Методы аутентификации AUTH_NONE, AUTH_SYS и AUTH_DES рассмотрены в одном из следующих разделов. В ONC вместо AUTH_SYS используется AUTH_UNIX.

Клиент вызывает RPC-функцию с помощью функции *RPC_cls::call*. Аргументами функции *RPC_cls::call* являются номер RPC-процедуры, XDR-функция, преобразующая входные аргументы в формат XDR, адрес переменной, в которой содержатся входные аргументы, XDR-функция, преобразующая значение, возвращенное данной RPC-функцией (из формата XDR в формат локальной машины), и адрес переменной, содержащей значение, возвращенное RPC-функцией. При успешном выполнении функция *RPC_cls::call* возвращает RPC_SUCCESS, а в случае неудачи — ненулевое значение.

Статическая функция *RPC_cls::broadcast* обеспечивает передачу широковещательных RPC-запросов из клиентского процесса по сети всем серверным процессам. Подробно эта функция рассмотрена в разделе 12.8.

Приложения, в которых используются классы RPC, в различных коммерческих разновидностях UNIX должны компилироваться со следующими опциями:

Разновидность UNIX	Опции компиляции CC
Solaris 2.x	-DSYSV4 -lsocket -lnsl
Sun OS 4.1.x	-lnsl
HP-UX 9.0.x	Нет
IBM AIX 3.x и 4.x	-lrpcsvc
SCO 3.x	-lsocket

Эти опции компиляции показывают, какие системные библиотеки RPC нужно связывать с пользовательскими приложениями в разных UNIX-системах. Опция -DSYSV4 в системе Solaris 2.x необходима для того, чтобы вместо API ONC использовались API RPC UNIX System V.4.

Чтобы проиллюстрировать использование классов RPC, перепишем программу удаленной печати сообщений, приведенную в разделе 12.3. Обратите внимание на то, что новые клиентская и серверная программы проще предыдущих версий, в которых используется компилятор *rpcgen*.

Клиентская программа *msg_cls2.C* выглядит следующим образом:

```
/* Клиентская программа: использование низкоуровневых API RPC  */
#include "msg2.h"
#include "RPC.h"

int main(int argc, char* argv[])
{
    if (argc<3)    {
        cerr <<"usage:" <<argv[0]<<"host msg <nettype>\n";
        return 1;
    }

    /* Создать обработчик клиента для RPC-сервера */
    RPC_cls cl( argv[1], MSGPROG, MSGVER, argc>=4 ? argv[3]:"netpath");
    if (!cl.good()) return 1;

    /* Вызвать RPC-функцию printmsg. Возвращаемое значение
       устанавливается в res */
    int res;
    if (cl.call( PRINTMSG, (xdrproc_t)xdr_string, (caddr_t)&argv[2],
                   (xdrproc_t)xdr_int, (caddr_t)&res) != RPC_SUCCESS)
        return 3;

    /* Проверить возвращаемое значение RPC-функции */
    if (res!=0)
        cerr << "clnt: call printmsg fails\n";
    else  cout << "clnt: call printmsg succeeds\n";

    return 0;
}
```

Эта клиентская программа вызывается с двумя или тремя аргументами. Первый аргумент — хост-имя машины, на которой работает RPC-сервер. Это может быть имя локальной машины, если клиент и сервер работают на одном компьютере. Второй аргумент — символьная строка сообщения, передаваемого на RPC-сервер для вывода на печать. Третий аргумент (необязательный) — это транспортный протокол, который должен использоваться клиентом для связи с сервером. Если третий аргумент не указан, по умолчанию принимается значение "*netpath*". Допустимые значения третьего аргумента такие же, как для аргумента *nettype* в API *clnt_create* (см. раздел 12.3.1).

Заголовок *msg2.h* определяется пользователем и содержит только объявление RPC-номеров программы, версии и процедуры для функции *printmsg*:

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#ifndef _MSG2_H_RPCGEN
#define _MSG2_H_RPCGEN
```



```
#include <rpc/rpc.h>

#define      MSGPROG ((unsigned long)(0x20000001))
#define      MSGVER ((unsigned long)(1))
#define      PRINTMSG ((unsigned long)(1))

#endif /* !_MSG_H_RPCGEN */
```

Программа RPC-сервера, соответствующая вышеупомянутой клиентской программе, называется *msg_svc2.C*:

```
/* программа-сервер: низкоуровневые API RPC */
/* вызов: msg_svc2 <механизм_транспортировки> */

#include <fstream.h>
#include <stdlib.h>
#include "msg2.h"
#include "RPC.h"

RPC_svc *svcp;

int printmsg( SVCXPRT* xtrp )
{
    int   res  = 0;
    char  *msg = 0;

    if (svcp->getargs( xtrp, (xdrproc_t)xdr_string,
                       ( caddr_t)&msg)!=RPC_SUCCESS)
        return -1;

    ofstream ofs("/dev/console");
    if (ofs)
        ofs << "server: '" << msg << "'\n";
    else res = -1;

    if (svcp->reply(xtrp, (xdrproc_t)xdr_int, (caddr_t)&res)!=RPC_SUCCESS)
        res = -2;

    return res;
}

int main(int argc, char* argv[])
{
    RPC_svc *svcp = new RPC_svc( MSGPROG, MSGVER, argc==2 ?
                                 argv[1] : "netpath");

    if (svcp && svcp->run_func( PRINTMSG, printmsg )) return 2;

    return 0; /* сюда программа переходит только в том случае,
                 если создание обработчика сервера завершается
                 неудачно */
}
```

Серверная программа вызывается либо без аргументов, либо со спецификацией *nettype*. Если аргумент *nettype* отсутствует, по умолчанию принимается значение "*netpath*".

Сначала серверный процесс вызывает функцию *RPC_svc::RPC_svc*, которая создает интерфейс к серверному процессу для RPC-функции с заданным номером программы, номером версии и значением *nettype*. Затем сервер вызывает функцию *RPC_svc::run_func*, которая регистрирует функцию *printmsg* как вызываемую клиентом RPC-функцию, после чего активизирует *svc_run* для отслеживания клиентских RPC-запросов.

Для обработки поступившего клиентского RPC-запроса функция *svc_run* вызывает функцию *RPC_svc::dispatch*, которая проверяет RPC-номер клиентской процедуры и аутентификацию клиента (если это необходимо). Затем вызывается затребованная RPC-функция.

Эта функция активизирует функцию *RPC_svc::getargs*, которая извлекает значения аргумента клиента. RPC-функция передает возвращенное значение клиенту с помощью функции *RPC_svc::reply*.

Последний фрагмент исходного кода, который необходимо включить в наш пример, представляет собой отдельный С-файл (*RPC.C*), содержащий определения статических переменных *RPC_svc::numProg* и *RPC_svc::progList*. Переменная *RPC_svc::progList* является указателем на динамический массив, в котором отслеживается каждая RPC-функция, соответствующая уникальной комбинации номера программы, номера версии и номера процедуры. Переменная *RPC_svc::numProg* содержит число достоверных элементов массива *RPC_svc::progList*.

Файл *RPC.C* имеет следующий вид:

```
#include "RPC.h"

int           RPC_svc::numProg = 0;
RPCPROG_INFO *RPC_svc::progList = 0;
```

Клиентская и серверная программы *printmsg* компилируются (в системе Solaris 2.x) и запускаются следующим образом:

```
% CC -DSYSV4 -c RPC.C
% CC -DSYSV4 msg_cls2.C PRC.o -o msg_cls2 -lsocket -lnsl
% CC -DSYSV4 msg_cls2.C PRC.o -o msg_cls2 -lsocket -lnsl
% msg_svc2 &
[135]
% msg_cls2 fruit "Hello RPC world"
clnt: call printmsg succeeds
```

В этом примере и клиентский, и серверный процессы выполняются на машине *fruit*. Серверная программа запускается в фоновом режиме, а клиентская — с аргументом *Hello RPC world*. После выполнения клиентской программы серверная программа выводит на системной консоли машины *fruit* сообщение *server: 'Hello RPC world'*.

Чтобы пользователи лучше разбирались в том, как работают классы RPC, в следующих разделах мы рассмотрим низкоуровневые API удаленных вызовов процедур.

12.5.1. Функция svc_create

Синтаксис функции *svc_create* имеет следующий вид:

```
#include <rpc/rpc.h>

int svc_create (void (*dispatch)(struct svc_req*, SVCXPRT*),
                u_long prognum, u_long versnum, char* nettype
```

Функция *svc_create* создает конечную точку транспортировки для заданного значения *nettype* в серверном процессе. Сервер контролирует все RPC-обращения к функции с указанными номерами программы и версии. Для каждого из этих RPC-запросов вызывается функция *dispatch*, которая отвечает на запрос.

Возможные значения аргумента *nettype* приведены в разделе 12.3.1.

Функция *dispatch* определяется пользователем и принимает два аргумента. Первый аргумент содержит информацию о клиентском RPC-вызове, которая используется для ответа на этот вызов. В частности, тип данных *struct svc_req* указывается в заголовке <rpc/svc.h> следующим образом:

```
struct svc_req
{
   u_long       rq_prog;       /* номер сервисной программы */
   u_long       rq_vers;       /* номер сервисного протокола */
   u_long       rq_proc;       /* требуемая процедура */
   struct opaque_auth rq_cred; /* необработанная аутентификационная
                                  информация */
   caddr_t      rq_clntcred;   /* обработанная аутентификационная
                                  информация только для чтения */
   struct_svcsxprt*  rq_xprt;  /* соответствующий метод
                                  транспортировки */
};
```

где поля *rp_prog*, *rq_vers* и *rq_proc* содержат номер программы, номер версии и номер процедуры RPC-функции, которую хочет вызвать клиент. Поля *rq_cred* и *rq_clntcred* содержат данные клиента, доступные функции *dispatch* для его аутентификации. Поле *rq_xprt* содержит информацию о механизме транспортировки, используемом клиентом; функция *dispatch*, как правило, это поле игнорирует.

Второй аргумент *svc_create* — указатель на конечную точку транспортировки. Он передается RPC-функции, которая использует его для получения значений аргументов функции от клиента. Кроме того, он служит для передачи клиенту возвращаемых значений.

При успешном выполнении эта функция возвращает ненулевой указатель на конечную точку транспортировки, а в случае неудачи — нуль.

В ONC для создания обработчиков RPC-серверов используются следующие функции:

```
#include <rpc/rpc.h>

SVCXPRT* svctcp_create (int svr_addr, const u_long sendbuf_size,
                                      const u_long recvbuf_size);
SVCXPRT* svcudp_create (int svr_addr);
```

Функции *svctcp_create* и *svcudp_create* — это TCP- и UDP-версии функции *svc_create*. Они организуют взаимодействие клиентского и серверного процессов с применением гнезд. Значение аргумента *svc_addr* — это номер порта гнезда, используемый RPC-сервером для связи со своими клиентами. Номер порта гнезда может быть задан как RPC_ANYSOCK, т.е. это может быть любой номер порта, присвоенный хост-системой.

Наконец, значения аргументов *sendbuf_size* и *recvbuf_size* задают размеры буферов для обмена данными между сервером и его клиентами.

12.5.2. Функция svc_run

Синтаксис функции *svc_run* имеет следующий вид:

```
#include <rpc/rpc.h>

void svc_run (void);
```

Эту функцию вызывает RPC-сервер. Она отслеживает клиентские RPC-запросы и для обслуживания поступившего запроса активизирует функцию *dispatch*, которая была зарегистрирована через API *svc_create*, *svc_tp_create* или *svc_tli_create*.

Функция *svc_run* не возвращает никаких значений.

12.5.3. Функция svc_getargs

Функция *svc_getargs* имеет следующий синтаксис:

```
#include <rpc/rpc.h>

boot_t svc_getargs (SVCXPRT* xprt, xdrproc_t* func, caddr_t argp);
```


Она вызывается RPC-функцией серверного процесса для того, чтобы выбрать аргументы, переданные клиентским процессом. Аргумент *xprt* — это указатель на назначенный клиентскому процессу механизм транспортировки. Аргумент *argp* содержит адрес переменной, которой присваиваются значения аргументов клиента. Аргумент *func* представляет собой указатель на XDR-функцию, которая используется для десериализации значений аргументов клиента в формат данных хост-машины сервера.

При успешном выполнении эта функция возвращает значение TRUE, а в случае неудачи — FALSE.

12.5.4. Функция svc_sendreply

Синтаксис функции *svc_sendreply* имеет следующий вид:

```
#include <rpc/rpc.h>

boot_t svc_sendreply (SVCXPRT* xprt, xdrproc_t* func, caddr_t resultp);
```

Она вызывается RPC-функцией в серверном процессе для передачи возвращаемых значений в клиентский процесс. Аргумент *xprt* — это указатель на назначенный клиентскому процессу механизм транспортировки. Аргумент *resultp* содержит адрес переменной, в которую помещаются возвращаемые функцией значения. Аргумент *func* представляет собой указатель на XDR-функцию, которая используется для сериализации возвращаемого значения в формат XDR.

При успешном выполнении эта функция возвращает значение TRUE, а в случае неудачи — FALSE.

12.5.5. Функция clnt_create

Функция *clnt_create* имеет следующий синтаксис:

```
#include <rpc/rpc.h>

CLIENT* clnt_create (char* hostnm, u_long prognum, u_long versnum,
                                                   const char* nettype);
```

Данная функция создает указатель CLIENT* для связи с RPC-сервером. Аргумент *hostnm* — это имя компьютера, на котором работает RPC-сервер. Аргументы *prognum* и *versnum* идентифицируют RPC-сервер по номеру программы и номеру версии. Аргумент *nettype* показывает, какой транспортный протокол используется для связи с серверным процессом.

Возможные значения аргумента *nettype* приведены в разделе 12.3.1.

При неудачном завершении функция *clnt_create* возвращает NULL, а в случае успеха — ненулевой указатель на клиента. Если функция завершилась неудачно, пользователи могут с помощью API *clnt_pcreateerror* просмотреть на стандартном устройстве выводе более подробное сообщение об ошибке. Функция *clnt_pcreateerror* имеет следующий прототип:

```
void clnt_pcreateerror (const char* msg_prefix_string);
```

Аргумент *msg_prefix_string* — это определяемая пользователем строка сообщения, которую функция выводит вместе с сообщением об ошибке.

12.5.6. Функция clnt_call

Синтаксис функции *clnt_call* имеет следующий вид :

```
#include <rpc/rpc.h>

enum clnt_stat
      clnt_call (CLIENT* clntp, u_long funcnum, xdrproc_t argfunc,
                         caddr_t argp, xdrproc_t resfunc, caddr_t resp,
                         struct timeval timv);
```

Она вызывается в клиентском процессе для вызова RPC-функции. Аргумент *clntp* — это указатель, полученный из API *clnt_create*, *clnt_tp_create* или *clnt_tli_create*. Аргумент *funcnum* — это номер процедуры RPC-функции. Аргумент *argfunc* представляет собой адрес XDR-функции, которая применяется для сериализации данных входных аргументов клиента в формат XDR перед передачей в RPC-функцию. Аргумент *resfunc* — это адрес XDR-функции, которая используется для десериализации возвращаемых значений RPC-функции в формат данных клиента. Аргумент *timv* задает продолжительность тайм-аута для данного вызова (в секундах или микросекундах).

При успешном выполнении функция *clnt_call* возвращает значение RPC_SUCCESS, а в случае неудачи — ненулевой код завершения. Если функция завершилась неудачно, пользователи могут с помощью API *clnt_perror* просмотреть на стандартном устройстве вывода подробное сообщение об ошибке. Функция *clnt_perror* имеет следующий прототип:

```
void clnt_perror (const CLIENT* clntp, const char* msg_prefix);
```

Аргумент *msg_prefix* — это определяемая пользователем строка сообщения, которую функция выводит вместе с сообщением об ошибке. Аргумент *clntp* является указателем, который идентифицирует вызывающий процесс.

12.6. Управление набором RPC-программ и версий

С помощью классов RPC можно создать серверный процесс, который управляет несколькими RPC-программами. Каждая программа может содержать одну и более версий набора RPC-функций. Чтобы разобраться в том, как это осуществляется, рассмотрим пример.

В этом примере серверная программа обслуживает две RPC-программы: PROG1NUM и PROG2NUM. Программа PROG1NUM содержит две версии (VERS1NUM и VERS2NUM) RPC-функции с номером процедуры PROC1-NUM и одну версию (VERS1NUM) RPC-функции с номером процедуры PROC2NUM. Вторая программа (PROG2NUM) состоит из одной RPC-функции с номером версии VERS2NUM и номером процедуры PROC2NUM. Объявления этих номеров программ, версий и процедур содержатся в заголовке *test.h*:

```
#ifndef TEST_H
#define TEST_H

#define PROG1NUM     0x20000010
#define PROG2NUM     0x20000015

#define VERS1NUM     0x1
#define VERS2NUM     0x2

#define PROC1NUM     0x1
#define PROC2NUM     0x2

#endif
```

Программа RPC-сервера называется *test_svc.C*:

```
#include "RPC.h"
#include "test.h"

/* RPC-функция: progno=1, vers=1, proc_no=1 */
int func1_1_1 (SVCXPRT* xprt)
{
    cerr <<"func1_1_1 called\n";
    svclp->reply(xprt, (xdrproc_t)xdr_void, 0);
    return RPC_SUCCESS;
}

/* RPC-функция: progno=1, vers=1, proc_no=2 */
int func1_1_2 (SVCXPRT* xprt)
{
    cerr <<"func1_1_2 called\n";
    svclp->reply(xprt, (xdrproc_t)xdr_void, 0);
    return RPC_SUCCESS;
}
```

```
/* RPC-функция: progno=1, vers=2, proc_no=1 */
int func1_2_1 (SVCXPRT* xprt)
 {
    cerr <<"func1_2_1 called\n";
    svc2p->reply(xprt, (xdrproc_t)xdr_void, 0);
    return RPC_SUCCESS;
}

/* RPC-функция: progno=2, vers=1, proc_no=1 */
int func2_1_1 (SVCXPRT* xprt)
{
    cerr <<"func2_1_1 called\n";
    svc3p->reply(xprt, (xdrproc_t)xdr_void, 0);
    return RPC_SUCCESS;
}

/* синтаксис: test_rpc [-s] [<nettype>] */
int main(int argc, char* argv[])
{

    char* nettype = (argc>1) ? argv[1] : "netpath";

    /* создать обработчик сервера для progno=1, vers=1 */
    svc1p = new RPC_svc ( PROG1NUM, VERS1NUM, nettype );
    /* создать обработчик сервера для prog_no=1, vers=2 */
    svc2p = new RPC_svc ( PROG1NUM, VERS2NUM, nettype );
    /* создать обработчик сервера для prog_no=2, vers=1 */
    svc3p = new RPC_svc ( PROG2NUM, VERS1NUM, nettype );

    if (!svc1p->good() || !svc2p->good() || !svc3p->good()) {
        cerr << "create server handle(s) failed\n";
        return 1;
    }

    /* зарегистрировать RPC-функцию */
    svc1p->add_proc( PROC1NUM, func1_1_1 );
    svc1p->add_proc( PROC2NUM, func1_1_2 );
    svc2p->add_proc( PROC1NUM, func1_2_1 );
    svc3p->add_proc( PROC1NUM, func2_1_1 );

    /* ожидать RPC-запросы клиентов для всех серверов */
    RPC_svc::run();
    return 0;
}
```

Эта серверная программа берет из командной строки необязательный аргумент, который задает тип транспортного протокола. Если аргумента нет, по умолчанию принимается значение "*netpath*".

Сервер создает три объекта *RPC_svc* — по одному для каждой версии RPC-программы, которой он управляет:

RPC_svc	Управляемая программа	Управляемая версия
svc1p	PROG1NUM	VERS1NUM
svc2p	PROG1NUM	VERS2NUM
svc3p	PROG2NUM	VERS1NUM

Сервер использует созданные объекты для регистрации RPC-функции. Имя функции формируется следующим образом: сначала идет префикс *func*, затем номер программы, знак подчеркивания, номер версии, опять знак подчеркивания и, наконец, номер процедуры. Так, *func1_2_1* означает, что эта функция представляет собой версию 2 процедуры 1 RPC-программы 1.

После регистрации всех этих RPC-функций сервер вызывает функцию *RPC_svc::run* и ожидает клиентские RPC-запросы. При поступлении запроса активизируется функция *RPC_svc::dispatch*, которая, в свою очередь, вызывает одну из зарегистрированных RPC-функций (в соответствии с указанными клиентом номерами программы, версии и процедуры).

Имя программы-клиента в нашем примере — *test_cls.C*:

```
#include "RPC.h"
#include "test.h"

int main(int argc, char* argv[])
{
    if (argc < 2) {
        cerr << "usage:" << argv[0] << " <server-host> [<nettype>]\n";
        return 1;
    }
    char* nettype = (argc > 2) ? argv[2] : "netpath";

    while (1) { /* client */

       unsigned progid, progno, verno, procno;

       /* получить требуемые номер программы, номер версии и номер
          функции */
       do {
           cout << "Enter prog#, ver#, proc#: " << flush;
           cin >> progno >> verno >> procno;
           if (cin.good()) break;
           if (cin.eof()) return 0;
       } while (1);

       /* преобразовать пользовательский номер программы
          во внутренний номер */
       progid = (progno==1) ? PROG1NUM : PROG2NUM;

       RPC_cls *clsp = new RPC_cls ( argv[1], progid, verno, nettype);
```

```
    if (!clsp->good()) {
        cerr << "create client handle(s) failed\n";
        return 2;
    }

    /* вызвать затребованную пользователем RPC-функцию */
    if (clsp->call( procno, (xdrproc_t)xdr_void, 0,
                            (xdrproc_t)xdr_void, 0 )
            != RPC_SUCCESS)
        cerr << "client call RPC function fails\n";

    delete clsp;
  }
  return 0;
}
```

При вызове клиентской программы в качестве аргумента задается хост-имя сервера и, по желанию, значение *nettype*. Если аргумент *nettype* не указан, по умолчанию принимается "*netpath*".

Клиентская программа выполняется в диалоговом режиме и предлагает пользователю ввести номера программы, версии и процедуры каждой вызываемой RPC-функции. Клиентский процесс создает объект *RPC_cls* для каждого полученного таким образом набора номеров и вызывает через него затребованную функцию. Завершается клиентский процесс, когда в потоке ввода встречается признак конца файла.

Эти программы компилируются и запускаются следующим образом (в нашем примере и клиент, и сервер выполняются на машине *fruit*):

```
% CC -DSYSV4 test_cls.C RPC.C -o test_cls -lsocket -lnsl
% CC -DSYSV4 test_svc.C RPC.C -o test_svc -lsocket -lnsl
% tets_svc &
[1235]
% test_cls fruit
Enter prog#, ver#, proc#: 1 1 1
*** func1_1_1 called
Enter prog#, ver#, proc#: 1 1 2
*** func1_1_2 called
Enter prog#, ver#, proc#: 1 2 1
*** func1_2_1 called
Enter prog#, ver#, proc#: 2 1 1
*** func2_1_1 called
Enter prog#, ver#, proc#: 1 1 0
Enter prog#, ver#, proc#: 4 1 2
fruit: RPC: Procedure unavailable
client call RPC function fais
Enter prog#, ver#, proc#: ^D
```

В этом случае RPC-функции вызывались в таком порядке: *func1_1_1*, *func1_1_2*, *func1_2_1* и *func2_1_1*. Вводом номеров 1 1 0 пользователь заставляет клиентскую программу запрашивать у RPC-сервера программу PROG1NUM (версия VERS1NUM). Для этой операции не задан вывод ответного сообщения. Ввод номеров 4 1 2 — это попытка вызывать несуществующую

RPC-функцию, поэтому и функция *RPC_cls::call*, и клиентская программа *test_cls.C* выдали сообщения об ошибке.

12.7. Аутентификация

Доступ к некоторым RPC-услугам разрешен только определенным пользователям. Поэтому клиентские процессы должны идентифицировать себя серверам, иначе затребованные RPC-функции вызываться не будут. В UNIX-системах есть несколько базовых методов аутентификации, но пользователи могут применять и собственные методы.

К встроенным в UNIX System V.4 методам RPC-аутентификации относятся AUTH_NONE, AUTH_SYS, AUTH_SHORT и AUTH_DES. В ONC применяются следующие методы: AUTH_NONE, AUTH_UNIX (эквивалент AUTH_SYS) и AUTH_DES. Рассмотрим эти методы подробнее.

Для аутентификации в данных аргумента *struct svc_req*, передаваемого клиентом функции-диспетчеру RPC-сервера, указываются номера целевой функции (номер программы, номер версии и номер процедуры) и аутентификационные данные клиента. Тип *struct svc_req* объявляется следующим образом:

```
struct svc_req
{
   u_long              rq_prog;     /* номер сервисной программы */
   u_long              rq_vers;     /* номер сервисного протокола */
   u_long              rq_proc;     /* требуемая процедура */
   struct opaque_auth  rq_cred;     /* необработанная
                                       аутентификационная
                                       информация */
   caddr_t             rq_clntcred; /* обработанная
                                       аутентификационная
                                       информация только для
                                       чтения */
   struct_svcxprt*     rq_xprt;     /* соответствующий механизм
                                       транспортировки */
};
```

где тип данных *struct opaque_auth* объявляется в заголовке <rpc/auth.h> следующим образом:

```
struct opaque_auth
{
   enum_t        oa_flavor;        /* метод аутентификации */
   caddr_t       oa_base;          /* указатель на специальные
                                     аутентификационные данные */
   u_int         oa_length;        /* размер данных, указанных с
                                     помощью oa_base */
};
```

В поле *opaque_auth::oa_flavor* указывается метод аутентификации, используемый клиентом. Если значение этого аргумента равно AUTH_NONE,

AUTH_SYS, AUTH_SHORT или AUTH_DES, то значения полей *opaque_auth::oa_base* и *opaque_auth::oa_length* никакой роли не играют. В поле *svc_req::rq_clntcred* указывается запись, в которой содержатся соответствующие аутентификационные данные.

Если в поле *opaque_auth::oa_flavor* содержится одно из значений AUTH_xxx, то в поле *opaque_auth::oa_base* — указатель на определяемую пользователем запись аутентификационных данных, а в поле *opaque_auth::oa_length* — размер записи, указанной в поле *opaque_auth::oa_base*.

Дальше мы рассмотрим методы RPC-аутентификации, встроенные в UNIX-системы. На их основе пользователи могут создавать собственные методы аутентификации.

12.7.1. Метод AUTH_NONE

Этот метод RPC-аутентификации применяется в UNIX System V по умолчанию и в действительности никакой аутентификации не выполняет. Клиент может установить этот метод явно, вызвав API *authnone_create*:

```
CLIENT* clntp = clnt_create(...);
if (clntp) {
    clntp->cl_auth = authnone_create();
    clnt_call(clntp, ...);
}
```

После этого для всех функций-диспетчеров RPC, вызываемых этим клиентом, значение *svc_req::rq_cred.oa_flavor* будет равно AUTH_NONE. Эти функции должны игнорировать значения *svc_req::rq_cred.oa_base*, *svc_req::rq_cred.oa_length* и *svc_req::rq_clntcred*.

12.7.2. Метод AUTH_SYS (или AUTH_UNIX)

В этом методе применяется управление доступом процессов ОС UNIX, т.е. для аутентификации клиентов используются UID и GID процессов. Клиент может установить этот метод явно, вызвав API *authsys_create_default*:

```
CLIENT* clntp = clnt_create(...);
if (clntp) {
    clntp->cl_auth = authsys_create_default();
    clnt_call(clntp, ...);
}
```

После этого для всех вызываемых клиентом функций-диспетчеров RPC значение *svc_req::rq_cred.oa_flavor* будет равно AUTH_SYS, а поле *svc_req::rq_clntcred* будет указывать на запись данных со следующей структурой:

```
struct authsys_parms
{
   u_long      aup_time;        /* время создания аутент. данных */
   char*       aup_machname;    /* имя машины клиента */
   uid_t       aup_uid;         /* эффективный UID клиента */
   gid_t       aup_guid;        /* эффективный GID клиента */
   u_int       aip_len;         /* число элементов в aup_gids */
   gid_t*      aup_gids;        /* дополнительные GID клиента */
};
```


Ниже приведен пример серверной функции-диспетчера, которая выполняет аутентификацию клиента:

```
int dispatch( struct svc_reg* rqstp, SVCXPRT* xtrp )
{
    struct authsys_parms* ptr;
    switch (rqstp->0a_flavor) {
      case AUTH_NONE:
        break;
      case AUTH_SYS:
        ptr = (struct authsys_parms*)rqst->rq_clntcred;
        if (ptr->aup_uid != 0 {
            svcerr_systemerr(xtrp);
            return;
        }
      case AUTH_DES:
        ...
        break;
      default:
        svcerr_weakauth( xtrp );
        return;
    }
    /* выполнить или вызвать реальную RPC-функцию */
}
```

В этом примере RPC-сервер пропускает проверку полномочий клиента, если в *rqstp->rq_cred.oa_flavor* указан метод аутентификации AUTH_NONE. Если же клиент выбрал метод AUTH_SYS, сервер проверяет, является ли действующий UID клиента идентификатором привилегированного пользователя (через аргумент *rqstp->rq_clntcred.aup_uid*). Если он таковым не является, сервер вызывает API *svcerr_systemerr* для вывода системного сообщения об ошибке. Это просто пример, и реальные пользовательские приложения могут проверять UID и/или GID клиентов так, как считают нужным.

Если для аутентификации клиента не задан ни один из стандартных системных методов, сервер вызывает API *svcerr_weakauth*, с помощью которого клиенту посылается сообщение об ошибке аутентификации "unsupported".

API *svcerr_weakauth* и *svcerr_systemerr* имеют следующие прототипы функций:

```
void svcerr_weakauth ( const SVCXPRT* xtrp );
void svcerr_systemerr ( const SVCXPRT* xtrp );
```

Аргумент *xtrp* в обеих функциях является указателем на механизм транспортировки, используемый для связи с клиентским процессом. Это значение передается как второй аргумент в функцию-диспетчер RPC-сервера.

В ONC вместо *authsys_create_default* используется API *authunix_create_default*, а константы AUTH_SYS и AUTH_SHORT заменены константой AUTH_UNIX. При этом во всех UNIX-системах применяется один и тот же базовый метод аутентификации, основанный на UID и GID процессов.

12.7.3. Метод AUTH_DES

Метод аутентификации AUTH_SYS прост в использовании, но не надежен, потому что не гарантирует уникальность параметров идентификации клиента (UID и GID) в масштабах Internet. Кроме того, перед RPC-вызовом клиент может запросто изменить данные *cl->cl_auth* так, чтобы "стать" кем-то другим. Для устранения этих недостатков был разработан метод AUTH_DES, обеспечивающий более надежную аутентификацию в RPC-приложениях.

Прежде чем воспользоваться методом AUTH_DES, клиентский процесс должен вызвать API *user2netname*, чтобы получить *сетевое имя* (*netname*), уникальность которого гарантируется в масштабах Internet. Это имя составляется следующим образом: перед именем домена клиентского процесса добавляется имя операционной системы процесса и действующий UID. Например, если процесс работает на UNIX-машине в домене *TJSys.com*, а действующий UID процесса — 125, то его сетевым именем будет *unix.125@TJSys.com*. Это имя уникально, потому что имя домена всегда уникально в Internet. Кроме того, в пределах домена каждый UID должен быть уникальным для всех машин, работающих под управлением одной операционной системы. Так, если домен *TJSys.com* содержит VMS-машины, на которых также есть пользователь с UID 125, то процесс, созданный этим пользователем, будет иметь сетевое имя *vms.125@TJSys.com*. Это имя отличает данный процесс от UNIX-процесса с тем же UID и именем домена.

В качестве альтернативы активизации функции *user2netname* процесс может вызвать API *host2netname* и получить сетевое имя для машины, на которой он выполняется. Это имя также является уникальным в Internet, однако оно обозначает не пользователя, а машину. Если процесс в приведенном выше примере работает на UNIX-машине с именем *fruit*, то функция *host2netname* возвратит сетевое имя *unix.fruit@TJSys.com*. Решение о том, каким API (*user2netname* или *host2netname*) пользоваться, зависит от того, на каком уровне RPC-приложения должны проводить аутентификацию — на уровне пользователей или машин (уровень выбирают пользователи).

API *user2netname* и *host2netname* имеют следующий синтаксис:

```
#include <rpc/rpc.h>

int user2netname (char netname[MAXNETNAMELEN+1],
                  uid_t eUID, const char* domain);
int host2netname (char netname[MAXNETNAMELEN+1],
                  const char* hostnm, const char* domain);
```


Первый аргумент обеих функций — это символьный буфер, минимальный размер которого составляет MAXNETNAME+1. Этот буфер предназначен для хранения возвращенного уникального имени процесса. Второй аргумент функции *user2netname* — действующий UID процесса, а второй аргумент функции *host2netname* — имя хост-машины. Третий аргумент в обеих функциях — имя домена. Если значение аргумента *domain* передается как NULL, подразумевается имя локального домена.

При успешном завершении эти функции возвращают 1, а в случае неудачи — 0.

Запись данных AUTH_DES в клиентском процессе создается с помощью API *authdes_seccreate*. Эта функция имеет следующий прототип:

```
#include <rpc/rpc.h>

int authdes_seccreate (char netname[MAXNETNAMELEN+1],
                       unsigned window, const char* time_host, const des_block* ckey);
```

Значение аргумента *netname* — либо сетевое имя вызывающего процесса, либо сетевое имя его хост-машины. Этот аргумент позволяет идентифицировать клиентский процесс.

Аргумент *window* задает промежуток времени (в секундах), по истечении которого аутентификационная информация клиента, установленная этим вызовом, теряет силу. Если RPC-сервер получит RPC-вызов от клиента, который идентифицирован более чем на *window* секунд позже, запрос будет отклонен.

Аргумент *time_host* задает имя машины, с которой будет получено значение текущего времени в момент аутентификации. Обычно это имя машины целевого RPC-сервера. Если этот аргумент задан как NULL, время клиента и сервера синхронизировать не нужно.

Аргумент *ckey* — это DES-ключ для шифрования мандата клиента (credential). Целевой сервер пользуется этим ключом для дешифровки мандата. Если этот аргумент задан как NULL, операционная система генерирует случайный ключ. Клиент может получить DES-ключ явно, вызвав API *key_gendes*.

В случае успешного выполнения API *authdes_seccreate* возвращает указатель AUTH*, который указывает на зашифрованную аутентификационную информацию клиента, а в случае неудачи — NULL.

API *key_gendes* создает DES-ключ для вызывающего процесса. Прототип этой функции выглядит следующим образом:

```
int key_gendes (des_block* ckey);
```

В качестве аргумента функции *key_gendes* используется адрес переменной типа *des_block*. Эта переменная предназначена для хранения сгенерированного DES-ключа. При успешном выполнении функция *key_gendes* возвращает 0, а в случае неудачи — -1.

На стороне RPC-сервера сервер может выбрать зашифрованную аутентификационную информацию клиента с помощью API *authdes_getucred* или *netname2host*. Первая из этих функций используется в том случае, если эта информация представляет собой сетевое имя пользователя (полученное из API *user2netname*). Этот API расшифровывает и возвращает серверу UID и GID клиента. Если аутентификационная информация клиента — сетевое имя машины (полученное из API *host2netname*), то вызывается API *netname2host*, который извлекает имя хост-машины клиента.

Ниже приведены прототипы этих функций:

```
#include <rpc/rpc.h>

int authdes_getucred (const struct authdes_cred* adc, uid_t* uid_p,
                      gid_t* gid_p, short* len_p, gid_t* gidArray);

int netname2host (const char* netname, char* hostname, int len);
```

Значение аргумента *adc* функции *authdes_getucred* получают из аргумента *rqstp* серверной функции-диспетчера. Это значение записано в поле *rqstp->rq_clntcred*, содержащем указатель на зашифрованную аутентификационную информацию клиента.

Аргументы *uid_p* и *gid_p* — это адреса переменных, содержащих возвращенные UID и GID клиента.

Аргумент *gidArray* — это адрес переменной, в которой хранится возвращаемый массив дополнительных GID, а аргумент *len* — это размер массива *hostname*.

Значение аргумента *netname* функции *netname2host* берется из аргумента *rqstp* серверной функции-диспетчера. Это значение содержится в поле *rqstp->rq_clntcred->adc_fullname.name*.

Аргумент *hostname* — это адрес символьного буфера, в котором хранится возвращенное имя хост-машины клиента. Аргумент *len* задает максимальный размер буфера, на который указывает аргумент *hostname*.

При успешном выполнении обе эти функции возвращают 1, а в случае неудачи — 0. Отметим, что в классах RPC, которые описаны в разделе 12.5, *RPC_cls::set_auth* может вызываться клиентским процессом для того, чтобы установить метод аутентификации AUTH_SYS или AUTH_DES. Во втором случае аутентификационная информация клиента состоит из его UID и GID.

На стороне сервера функция *RPC_svc::dispatch* проверяет каждого клиента, применяя установленный метод аутентификации. Если клиент задал метод AUTH_NONE, функция-диспетчер просто пропускает этап проверки.

В реальных защищенных RPC-приложениях это недопустимо. Пользователи могут изменить функцию-диспетчер таким образом, чтобы в случае задания клиентом метода AUTH_NONE она выставляла флаг ошибки и отказывалась выполнять затребованную RPC-функцию. Если клиент применяет метод AUTH_DES, функция-диспетчер вызывает API *authdes_getucred* для извлечения UID и GID клиента. Это допустимо в том случае, когда для создания мандата клиента *RPC_cls::set_auth* вызывает только API *user2netname* (а не *host2netname*). Если же в пользовательском приложении для этой цели используется функция *host2netname* и/или *user2netname*, то функцию *RPC_svc::dispatch* необходимо соответствующим образом откорректировать.

12.7.4. Получение списка файлов каталога с применением аутентификации

Вернемся к примеру, рассмотренному в разделе 12.3.3, и перепишем его, внеся следующие изменения:

- компилятор *rpcgen* заменим классами RPC;
- покажем, как клиентский процесс запрашивает серверный процесс;
- проиллюстрируем механизм RPC-аутентификации.

Файлы *RPC.h* и *RPC.C* приведены в разделе 12.5. Файл *scan2.h* создается вручную следующим образом:

```
#ifndef SCAN2_H
#define SCAN2_H

#include <rpc/rpc.h>

#ifdef __cplusplus
extern "C" {
#endif

#define MAXNLEN 255

typedef char *name_t;

typedef struct arg_rec *argPtr;

struct arg_rec {
    name_t dir_name;
    int lflag;
};
typedef struct arg_rec arg_rec;

typedef struct dirinfo *infoiist;

 struct dirinfo {
    name_t   name;
    u_int    uid;
    long     modtime;
    infolist next;
```

```
};
typedef struct dirinfo dirinfo;

struct res {
    int errno;
    union {
        infolist list;
    } res_u;
};
typedef struct res res;

#define    SCANPROG ((unsigned long)(0x20000100))
#define    SCANVER ((unsigned long)(1))

#if defined(__STDC__) || defined(__cplusplus)
#define    SCANDIR ((unsigned long)(1))
extern res * scandir_1(argPtr *, CLIENT *);
extern res * scandir_1_svc(argPtr *, struct svc_req *);
extern int scanprog_1_freeresult(SVCXPRT *, xdrproc_t, caddr_t);

#else /* K&R C */
#define    SCANDIR ((unsigned long)(1))
extern res * scandir_1();
extern res * scandir_1_svc();
extern int scanprog_1_freeresult();
#endif /* K&R C */

/* XDR-функции */

#if defined(__STDC__) || defined(__cplusplus)
extern bool_t xdr_name_t(XDR *, name_t*);
extern bool_t xdr_argPtr(XDR *, argPtr*);
extern bool_t xdr_arg_rec(XDR *, arg_rec*);
extern bool_t xdr_infolist(XDR *, infolist*);
extern bool_t xdr_dirinfo(XDR *, dirinfo*);
extern bool_t xdr_res(XDR *, res*);

#else /* K&R C */
extern bool_t xdr_name_t();
extern bool_t xdr_argPtr();
extern bool_t xdr_arg_rec();
extern bool_t xdr_infolist();
extern bool_t xdr_dirinfo();
extern bool_t xdr_res();

#endif /* K&R C */

#ifdef __cplusplus
)
#endif

#endif /* !SCAN2_H */
```


Клиентская программа *scan_cls2.C* выглядит следующим образом:

```
#include <errno.h>
#include "scan2.h"
#include "RPC.h"

//extern int errno;

int main( int argc, char* argv[])
{
    static res result;
    infolist nl;

    if (argc<3) {
        cerr << "usage: " << argv[0] << " host direcoty [<long>]\n";
        return 1;
    }

    /* создать обработчик RPC-клиента */
    /*RPC_cls cl( argv[1], SCANPROG, SCANVER, "netpath");*/
    RPC_cls cl( argv[1], SCANPROG, SCANVER, "tcp");
    if (!cl.good()) return 1;

    /* установить метод аутентификации */
    cl.set_auth( AUTH_DES );

    /* запросить RPC-сервер, чтобы проверить, что он работает */
    if (cl.call( 0, (xdrproc_t)xdr_void, 0, (xdrproc_t)xdr_void,
                                                0 ) == RPC_SUCCESS)
       cout << "Prog " << SCANPROG << " (version " << SCANVER << ")
                                            is alive\n";
    else {
    cerr << "Prog " << SCANPROG << " version " << SCANVER << " is dead!\n";
    return 2;
    }

/* выделить память для хранения возвращаемого списка файлов каталога */
    struct arg_rec *iarg = (struct arg_rec*)malloc(sizeof(struct arg_rec));

    iarg->dir_name = argv[2];  // установить имя удаленного каталога
    iarg->lflag = 0; // установить флаг вывода подробной информации
    if (argc==4 && sscanf(argv[3],"%u",&(iarg->lflag))!=1) {
        fprintf(stderr,"Invalid argument: '%s'\n", argv[3]);
        return 3;
    }

    /* вызвать RPC-функцию */
    if (cl.call( SCANDIR, (xdrproc_t)xdr_argPtr, (caddr_t)&iarg,
            (xdrproc_t)xdr_res, (caddr_t)&result) != RPC_SUCCESS)
    {
        cerr << "client: call RPC fails\n";
        return 4;
    }
```

```
    /* RPC-вызов завершен. Проверить код возврата функции */
    if (result.errno) {
        errno = result.errno;
        perror(iarg->dir_name);
        return 5;
    }
    /* RPC-функция завершается успешно. Вывести список файлов
       удаленного каталога */
    for (nl=result.res_u.list; nl; nl=nl->next) {
        if (iarg->lflag)
            cout << "..." << nl->name << ", uid=" << nl->uid << ", mtime="
                << ctime(&nl->modtime) << endl;
        else cout << "...'" <<  nl->name << "'\n";
    }
    return 0;
}
```

Программа-клиент вызывается с хост-именем сервера, именем удаленного каталога и, по желанию, с целочисленным флагом. Имя удаленного каталога — это имя каталога, содержимое которого должна возвратить данная RPC-функция. Необязательный целочисленный флаг задает формат, в котором должен быть представлен список файлов: в развернутом формате (*lflag* = 1) или только с именами файлов (*lflag* = 0).

Программа-клиент, вызванная с правильными аргументами, создает объект *RPC_cls* для соединения с сервером при помощи функции-конструктора *RPC_cls::RPC_cls*. RPC-сервер идентифицируется константами SCANPROG, SCANVER и SCANFUNC (номер программы, номер версии и номер процедуры).

После создания клиентского объекта *RPC_cls* клиентский процесс вызывает функцию *RPC_cls::set*, которая генерирует аутентификационную информацию клиента методом AUTH_DES. Эта функция может создавать мандаты, используя AUTH_NONE, AUTH_SYS или AUTH_DES, и скрывать все низкоуровневые API RPC-аутентификации от пользователей.

Затем клиент вызывает RPC-сервер с номером процедуры 0, чтобы проверить, работает ли сервер. Если функция *RPC_cls::call* завершается неудачно, клиентский процесс выводит соответствующее сообщение об ошибке и завершает свою работу; в противном случае его выполнение продолжается.

Клиент выделяет динамическую память для переменной *iarg*, в которой хранится входной аргумент RPC-функции: имя удаленного каталога и целочисленный флаг. Клиент вызывает RPC-функцию через функцию *RPC_cls::call* и указывает, что возвращаемое ею значение должно быть помещено в переменную *result*. Кроме того, XDR-функциями для входного аргумента и возвращаемого значения являются пользовательские функции *xdr_argPtr* и *xdr_res* соответственно.

Если RPC-функция возвращает код успешного завершения, клиентская программа выводит на экран возвращенное значение функции (перечень

содержимого удаленного каталога). Отметим, что если флаг формата отображения содержимого каталога равен 1, то информация о каждом файле включает его имя, UID и время последней модификации. Если же флаг формата равен 0, то показывается только имя файла.

Программа-сервер, которая выводит содержимое удаленного каталога, находится в файле *scan_svc2.C*:

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <string.h>
#include <malloc.h>
#include <sys/stat.h>
#include "scan2.h"
#include "RPC.h"

extern int errno;

static RPC_svc *svcp = 0;

/* RPC-функция */
int scandir( SVCXPRT* xtrp )
{
    DIR *dirp;
    struct dirent *d;
    infolist  nl, *nlp;
    struct stat statv;
    res res;
    argPtr darg = 0;

    /* получить от клиента аргумент функции */
    if (svcp->getargs( xtrp, (xdrproc_t)xdr_argPtr,
                    (caddr_t)&darg)!=RPC_SUCCESS)
        return -1;

    cerr << "server: get dir: '" << darg->dir_name << "', lflag="
                                << darg->lflag << endl;

    /* начать просмотр затребованного каталога */
    if (!(dirp = opendir(darg->dir_name))) {
        res.errno = errno;
        (void)svcp->reply(xtrp, (xdrproc_t)xdr_res, (caddr_t)&res);
        return -2;
    }
    /* освободить память, выделенную в предыдущем RPC-вызове */
    xdr_free((xdrproc_t)xdr_res, (char*)&res);
    /* записать информацию о файлах в res как возвращаемые значения */
    nlp = &res.res_u.list;
    while (d=readdir(dirp)) {
        nl = *nlp = (infolist)malloc(sizeof(struct dirinfo));
        nl->name = strdup(d->d_name);
```

```
        nlp = &nl->next;
        if (darg->lflag) {
            char pathnm[256];
            sprintf(pathnm,"%s/%s",darg->dir_name,d->d_name);
            if (!stat(pathnm,&statv)) {
                nl->uid = statv.st_uid;
                nl->modtime = statv.st_mtime;
            }
        }
    }
    *nlp = 0;
    res.errno = 0;
    closedir(dirp);
    /* передать список содержимого каталога клиенту */
    if (svcp->reply(xtrp, (xdrproc_t)xdr_res, (caddr_t)&res)!=RPC_SUCCESS)
        return -2;

    return RPC_SUCCESS;
}

/* главная функция RPC-сервера */
int main(int argc, char* argv[]);
{
    svcp = new RPC_svc( SCANPROG, SCANVER, "netpath");
    if (svcp->run_func( SCANDIR, scandir )) return 1;
    return 0;   /* здесь не должно быть выхода из программы */
}
```

Приведенная программа создает RPC-сервер, который выводит содержимое каталога. Аргументы командной строки для вызова программы не требуются.

Данный процесс создает с помощью функции-конструктора *RPC_svc::RPC_svc* объект *RPC_svc*. После этого для регистрации RPC-функции *scandir* в объекте *RPC_svc* сервер вызывает функцию *RPC_svc::run_func* и ожидает поступления клиентских RPC-запросов.

Получив запрос, сервер вызывает функцию *RPC_svc::dispatch*, которая прежде всего проверяет, равен ли номер затребованной RPC-процедуры нулю. Если клиент хочет только выяснить, готов ли сервер к работе, эта функция возвращает значение NULL, в чем и состоит ответ сервера на такой запрос.

Если же это не просто проверка готовности сервера к работе, функция-диспетчер проверяет запрос клиента по заданному методу аутентификации. Получив отрицательный результат, функция-диспетчер выводит сообщение о системной RPC-ошибке и завершает работу. Текущая функция *RPC_svc::dispatch* принимает только тех клиентов, у которых UID либо равен нулю (привилегированный пользователь), либо совпадает с UID серверного процесса.

Установив достоверность аутентификационной информации клиента, функция-диспетчер вызывает затребованную RPC-функцию. В этом примере

единственная RPC-функция — это *printmsg*, которая выполняет следующие операции:

- получает от клиентского процесса аргументы;
- освобождает динамическую память, выделенную переменной *res* в предыдущем вызове;
- просматривает указанный каталог и помещает информацию обо всех его файлах в переменную *res*;
- передает переменную *res* как возвращаемое значение в вызывающую программу-клиент.

Последние компоненты кода в этом примере — это XDR-функции для пользовательских типов данных (например, *struct arg_rec*, *argPtr*, *infolist* и т.д.). Эти XDR-функции определяются в файле *scan2_xdr.c*:

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "scan2.h"

/* XDR-функция для данных типа name_t */
bool_t xdr_name_t(register XDR *xdrs, name_t *objp)
{

    register long *buf;

    if (!xdr_string(xdrs, objp, MAXNLEN))
        return (FALSE);
    return (TRUE);
}

/* XDR-функция для данных типа argPtr */
bool_t xdr_argPtr(register XDR *xdrs, argPtr *objp)
{

    register long *buf;

    if (!xdr_pointer(xdrs, (char **)objp, sizeof (struct arg_rec),
                                    (xdrproc_t) xdr_arg_rec))
        return (FALSE);
    return (TRUE);
}

/* XDR-функция для данных типа arg_rec */
bool_t xdr_arg_rec(register XDR *xdrs, arg_rec *objp)
{

    register long *buf;

    if (!xdr_name_t(xdrs, &objp->dir_name))
        return (FALSE);
```

```
    if (!xdr_int(xdrs, &objp->lflag))
        return (FALSE);
    return (TRUE);
}

/* XDR-функция для данных типа infolist */
bool_t xdr_infolist(register XDR *xdrs, infolist *objp)
{

    register long *buf;

    if (!xdr_pointer(xdrs, (char **)objp, sizeof (struct dirinfo),
                                  (xdrproc_t) xdr_dirinfo))
        return (FALSE);
    return (TRUE);
)

/* XDR-функция для данных типа dirinfo */
bool_t xdr_dirinfo(register XDR *xdrs, dirinfo *objp)
{

    register long *buf;

    if (!xdr_name_t(xdrs, &objp->name))
        return (FALSE);
    if (!xdr_u_int(xdrs, &objp->uid))
        return (FALSE);
    if (!xdr_long(xdrs, &objp->modtime))
        return (FALSE);
    if (!xdr_infolist(xdrs, &objp->next))
        return (FALSE);
    return (TRUE);
}

/* XDR-функция для данных типа res */
bool_t xdr_res(register XDR *xdrs, res *objp)
{

    register long *buf;

    lf (!xdr_int(xdrs, &objp->errno))
        return (FALSE);
    switch (objp->errno) {
    case 0:
        if (!xdr_infolist(xdrs, &objp->res_u.list))
            return (FALSE);
        break;
    }
    return (TRUE);
)
```

Эти XDR-функции особых пояснений не требуют. Их можно генерировать вручную или с помощью *rpcgen*. В последнем случае пользователи

должны объявить свои типы данных в файле *rpcgen.x*, а там, где это необходимо, использовать типы данных, характерные для *rpcgen*, например *string*.

Приведенные выше клиентская и серверная программы создаются следующими командами shell:

```
% CC -c scan2_xdr.c RPC.C
% CC -DSYSV4 -o scan_svc2 scan_svc2.C scan2_xdr.o \
          RPC.o -lsocket  -lnsl
% CC -DSYSV4 -o scan_cls2 scan_cls2.C scan2_xdr.o \
          RPC.o -lsocket  -lnsl
```

Ниже приведены примерные результаты запуска этих программ. Серверная программа выполняется на машине *fruit*, а клиентскую можно запускать на любой машине, соединенной с *fruit*:

```
% scan_svc2 &
% scan_cls2 fruit .
....
....
...scan_cls2.C
...scan_svc2.C
...RPC.C
...RPC.h
...scan2_xdr.c
...scan2.h
...scan_svc2
...sca n_cls2
Prog 536871168 (version 1) is alive
```

12.8. Широковещательный режим RPC

Некоторые RPC-запросы могут потребовать ответа от всех серверов в сети, предоставляющих затребованные услуги. Допустим, например, что клиентскому процессу надо установить системное время на всех машинах в локальной сети. Предположим, что на каждой машине работает RPC-сервер, действующий UID которого соответствует UID привилегированного пользователя. Клиентский процесс одним RPC-вызовом передает новое системное время в широковещательном режиме всем этим серверам, и каждый сервер соответствующим образом корректирует системное время.

Чтобы использовать широковещательный режим RPC, процесс может задействовать функцию-член *RPC_cls::broadcast*. Это статическая функция, перед выполнением которой не требуется создавать объект *RPC_cls*. Эта функция, в свою очередь, вызывает API *rpc_broadcast*, с помощью которого и реализуется широковещательная передача. Прототип этого API выглядит следующим образом:

```
#include <rpc/rpc.h>

enum clnt_stat rpc_broadcast (unsigned prognum, unsigned versnum,
    unsigned funcnum, xdrproc_t argfunc, caddr_t argp, xdrproc_t resfunc,
    caddr_t resp, resultproc_t callme, char* nettype);
```

Аргументы *prognum*, *versnum* и *funcnum* — это номера вызываемой RPC-функции.

Аргумент *argfunc* представляет собой адрес XDR-функции, используемой для сериализации и десериализации аргумента RPC-функции, заданного как *argp*. Аргумент *resfunc* — это адрес XDR-функции, используемой для сериализации и десериализации возвращаемого значения RPC-функции, которое помещается в аргумент *resp*.

Аргумент *nettype* задает способ передачи широковещательного RPC-вызова. Это должен быть транспортный протокол без установления соединения, например UDP. По умолчанию аргумент *nettype* в функции *RPC_cls::broadcast* имеет значение "*datagram_v*". Это значит, что может использоваться любой протокол на основе дейтаграмм, который в файле */etc/netconfig* указан как "*visible*". Для функции *rpc_broadcast* характерны еще два ограничения: во-первых, широковещательный запрос по размеру не может превышать установленное для его хост-машины значение параметра MTU (для локальных сетей на базе Ethernet MTU равен 1500 байтам); во-вторых, на широковещательные RPC-запросы могут отвечать только серверы, зарегистрированные в демоне *rpcbind* (т.е. серверы должны быть созданы с помощью API *svc_create* или *svc_tp_create*).

Аргумент *callme* — это определяемая пользователем функция, которая вызывается для каждого ответа RPC-сервера. Она имеет следующий прототип:

```
int callme (caddr_t resp, struct netbuf* server_addr, struct netconf* nconf);
```

где аргумент *resp* принимает то же значение *resp*, которое указано в вызове *rpc_broadcast*. Это адрес определенной в клиентском процессе переменной, которая содержит значение, возвращаемое сервером. Аргумент *server_addr* содержит адрес отвечающего сервера, а аргумент *nconf* — информацию о методе транспортировки, используемом сервером.

Активизированная функция *rpc_broadcast* блокирует вызывающий процесс, заставляя его ожидать ответов от серверов. Для обслуживания каждого такого ответа эта функция вызывает функцию *callme*. Если функция *callme* возвращает значение 0, то *rpc_broadcast* ожидает следующего ответа. Если функция *callme* возвращает ненулевое значение, то функция *rpc_broadcast* завершается и передает управление вызывающему процессу.

Функция *rpc_broadcast* возвращает значение RPC_TIMEOUT, если она ожидала и пробовала передать широковещательный запрос несколько раз,

но не получила никакого ответа. Если функция *callme* возвратила TRUE, то функция *rpc_broadcast* возвращает значение RPC_SUCCESS; в противном случае она возвращает ненулевое значение, свидетельствующее об ошибке.

Идентификацию вызывающего процесса для всех серверных процессов, которые принимают широковещательный запрос, функция *rpc_broadcast* выполняет по методу AUTH_SYS.

В ONC функцию *rpc_broadcast* заменяет API *clnt_broadcast*. Эти функции имеют почти одинаковые сигнатуры и возвращаемые значения, но в *clnt_broadcast* не используется аргумент *nettype*.

```
#include <rpc/rpc.h>

enum clnt_stat clnt_broadcast (unsigned prognum, unsigned versnum,
                    unsigned funcnum, xdrproc_t argfunc, caddr_t argp,
                    xdrproc_t resfunc, caddr_t resp, resultproc_t callme);
```

Ниже приведен прототип функции *callback* для API *clnt_broadcast*:

```
int callme (caddr_t resp, struct sockaddr_in* server_addr);
```

где аргумент *resp* принимает то же значение *resp*, которое указано в вызове *clnt_broadcast*. Аргумент *server_addr* содержит адрес отвечающего сервера. Его тип данных — указатель на структуру, в которой хранится адрес гнезда.

12.8.1. Пример широковещательной передачи RPC-запросов

Сейчас мы изменим представленную в разделе 12.5 программу *msg_cls2.C*, включив в нее средства широковещательного режима RPC (изменения касаются только программы-клиента). Новая программа называется *msg_cls3.C*:

```
/* программа-клиент: использование широковещательного режима для
   вывода сообщения на системную консоль сервера */
#include "msg2.h"
#include "RPC.h"

static int num_responses = 0;

/* вызов широковещательного режима клиента */
bool_t callme (caddr_t res_p, struct netbuf* addr, struct netconfig *nconf)
{
    num_responses++;        // число ответивших серверов

    if (res_p==0 || *((int*)res_p)!=0) {
        cerr << "clnt: call printmsg fails\n";
        return TRUE;        // прекратить широковещательную передачу
    }
    cout << "clnt: call printmsg succeeds\n";
```

```
    return FALSE;                    // ожидание ответов
}

/* клиентская главная функция */
int main(int argc, char* argv[])
{
    int res;
    if (argc<2) {
        cerr << "usage: " << argv[0] << " msg\n";
        return 1;
    }

    /* клиент посылает широковещательный запрос и ждет ответа */
    int rc = RPC_cls::broadcast( MSGPROG, MSGVER, PRINTMSG,
                        (resultproc_t)callme,(xdrproc_t)xdr_string,
                        (caddr_t)&argv[1],(xdrproc_t)xdr_int,
                        (caddr_t)&res);

    switch (rc) {
        case RPC_SUCCESS:     break;
        case RPC_TIMEDOUT:
            if (num_responses) break;
        default:
            cerr << "RPC broadcast failed\n";
            return 2;
    }
    cout << "RPC braodcast done. No. responses: " << num_responses << endl;
    return 0;
}
```

Новая клиентская программа *printmsg* передает один аргумент командной строки как сообщение в широковещательном режиме. Для этого используется функция *RPC_cls::broadcast*. При ее активизации клиентский процесс задает *callme* как функцию, которую функция *rpc_broadcast* должна вызывать для каждого ответа сервера. Аргумент и XDR-функция программы *printmsg*, а также переменная, содержащая возвращаемое значение и XDR-функцию программы *printmsg*, устанавливаются в вызове функции *RPC_cls::broadcast* так же, как в вызове функции *RPC_cls::call*.

Функция *callme* вызывается для каждого ответа на широковещательный запрос. Эта функция просто проверяет возвращаемое значение. Если сервер возвращает код неудачного завершения, то функция возвращает TRUE и широковещательная передача прекращается; в противном случае возвращается FALSE и прием ответов продолжается. Функция *callme* увеличивает на единицу значение глобальной переменной *num_responses*, в которой хранится число серверов, фактически ответивших на запрос.

После выполнения функции *RPC_cls::broadcast* клиентская программа проверяет код возврата этой функции. Если возвращено значение RPC_SUCCESS, значит, широковещательная передача была остановлена функцией *callme* и все нормально. Если же возвращен код RPC_TIMEOUT, то проверяется

значение переменной *num_responses*, по которому можно определить, ответил ли на запрос хоть один сервер. Нулевое значение свидетельствует о неудаче широковещательного RPC-запроса (при этом выставляется флаг ошибки). Ненулевое значение переменной *num_responses* соответствует успешной обработке широковещательного запроса. Функция *rpc_broadcast* завершает свою работу, поскольку все серверы ответили на запрос.

Ниже представлены результаты работы серверной программы *msg_svc2* (см. разд. 12.5) и новой клиентской программы *msg_cls3*, функционирующей в режиме широковещательных RCP-запросов:

```
% CC _DSYSV4 -o msg_cls3 msg_cls3.C RPC.C -lsocket -lnsl
% msg_cls3 "Testing RPC broadcast feature"
clnt: call printmsg succeeeds
clnt: call printmsg succeeeds
...
```

На системных консолях всех машин (на которых работает демон *msg_svc2*) выводится сообщение *Testing RPC broadcast feature*.

12.9. Обратный вызов RPC

В некоторых RCP-приложениях необходимо, чтобы сервер выполнял обратный вызов (call back) клиентского процесса через определенный промежуток времени. Клиентский процесс может в течение этого времени проводить какие-то другие операции. Например, клиентский процесс просит RPC-сервер выполнить некую трудоемкую функцию, но не хочет останавливаться и ждать, пока эта работа завершится. В этом случае клиент задает RPC-функцию, которую сервер сможет вызвать, когда будет готов передать результаты клиенту. Таким образом и клиент, и сервер могут параллельно выполнять полезную работу, что повышает общую эффективность системы.

Чтобы RPC-сервер мог выполнить обратный вызов, клиент должен указать для функции обратного вызова номер RPC-программы, номер версии и номер процедуры. В некотором смысле этот клиент исполняет роль и клиента, и сервера.

Давайте посмотрим, как это можно сделать. В приведенном ниже примере RPC-сервер предоставляет процессам в локальной сети услуги будильника. Клиентский процесс, которому нужна эта услуга, посылает серверу RCP-запрос и указывает следующую информацию:

- хост-имя машины клиентского процесса;
- номер программы, версии и процедуры RPC-функции обратного вызова клиента;
- интервал времени в секундах до сигнала будильника (alarm clock).

Получив запрос от клиента, RPC-сервер порождает дочерний процесс, чтобы установить сигнал будильника, который будет передан порожденному процессу по истечении заданного клиентом интервала времени. Чтобы

сообщить клиентскому процессу о том, что указанное время истекло, порожденный процесс активизирует функцию обратного вызова клиента, после чего завершается. Во время всех этих операций RPC-сервер непрерывно отслеживает другие возможные запросы сервиса будильника. В течение заданного интервала времени до получения сигнала будильника исходный клиентский процесс может выполнять другие задачи.

Заголовочный файл *aclock.h* совместно используется клиентской и серверной программами:

```
#ifndef ACLOCK_H
#define ACLOCK_H

#include <rpc/rpc.h>

#define    MAXNLEN 255

typedef char *name_t;

/* информация для обратного вызова клиента RPC-сервером */
struct arg_rec {
    name_t hostname;        // хост-имя машины клиента
    u_long prognum;         // номер программы RPC-функции клиента
    u_long versnum;         // номер версии RPC-функции клиента
    u_long procnum;         // номер процедуры RPC-функции клиента
    u_long atime;           // временной интервал для будильника
);

#define CLNTVERNUM  1
#define CLNTPROCNUM 1

#define ACLKPROG ((unsigned long)(0x20000100))
#define ACLKVER ((unsigned long)(1))
#define ACLKPROC ((unsigned long)(1))

/* XDR-функции для преобразования данных обратного вызова клиента */
extern bool_t xdr_name_t(XDR *, name_t*);
extern bool_t xdr_arg_rec(XDR *, arg_rec*);

#endif /* !ACLOCK_H */
```

Программа RPC-сервера *aclk_svc.C* выглядит следующим образом:

```
#include <signal.h>
#include "aclock.h"
#include "RPC.h"

RPC_svc *svcp;                      // обработчик RPC-сервера
static struct arg_rec argRec;       // содержит информацию для
                                    // обратного вызова клиента

/* выполнить обратный вызов RPC-функции клиента*/
void call_client( int signum )
{
    u_long x = alarm(0);            //остаток интервала будильника
    RPC_cls cls( argRec.hostname, argRec.prognum, argRec.versnum, "netpath");
    if (!cls.good()) {
        cerr <<"call_client: create RPC_cls object failed\n";
        exit(1);
    }
    if (cls.call( argRec.procnum, (xdrproc_t)xdr_u_long, (caddr_t)&x,
                  (xdrproc_t)xdr_void, 0 )!=RPC_SUCCESS)
        cerr << "call_client: call client failed\n";

    exit(0); /* уничтожить порожденный процесс */
}
/* RPC-функция сервера. Вызывается клиентом для настройки сервиса
   будильника */
int set_alarm( SVCXPRT* xtrp )
{
/* получить информацию о клиенте: хост-имя, номер программы,
   номер версии и номер процедуры функции обратного вызова RPC */
    if (svcp->getargs( xtrp, (xdrproc_t)xdr_arg_rec,
                       (caddr_t)&argRec)!=RPC_SUCCESS)
        return -1;

    /* возвратить клиенту полученные значения */
    if (svcp->reply( xtrp, (xdrproc_t)xdr_void,
(caddr_t)0)!=RPC_SUCCESS)
    {
        cerr << "printmsg: sendreply failed\n";
        return -1;
    }

    /* создать порожденный процесс для работы с данным клиентом */
    switch (fork()) {
        case -1:    perror("can't fork");
                    break;
        case 0:     /* порожденный процесс */
                    alarm(argRec.atime);
                    signal(SIGALRM, call_client);
                    pause();        // ожидать истечения
                                    // интервала времени до
                                    // сигнала будильника
    }
    /* родительский процесс */
    return RPC_SUCCESS;
}

int main(int argc, char* argv[])
{
    /* создать обработчик сервера для ожидания RPC-вызовов функции
       set_alarm */
    RPC_svc *svcp = new RPC_svc( ACLKPROG, ACLKVER, argc==2 ?
                                 argv[1] : "netpath");
```

```
    /* зарегистрировать RPC-функцию и ожидать RPC-вызовы */
    if (svcp->run_func( ACLKPROC, set_alarm )) return 1;
    return 1; /* сюда серверный процесс не должен попасть никогда */
}
```

Серверная программа начинает работу с создания объекта *RPC_svc* для инициирования RPC-функции *set_alarm*. Номера программы, версии и процедуры этой RPC-функции — ACLKPROG, ACLKVER и ACLKPROC соответственно. Вызвав функцию *RPC_svc::run*, сервер ожидает поступления клиентских RPC-запросов.

Получив клиентский RPC-запрос, сервер вызывает RPC-функцию *set_alarm*, которая, в свою очередь, активизирует функцию *RPC_svc::getargs* для извлечения информации об обратном вызове RPC-функции. Эта информация хранится в переменной *argRec*. После успешного выполнения вызова *RPC_svc::getargs* сервер активизирует функцию *RPC_svc::reply*, которая возвращает клиенту полученные значения. Обратный вызов завершается, и клиент может перейти к выполнению других операций.

После вызова *RPC_svc::reply* сервер порождает дочерний процесс для обслуживания данного клиента, а родительский процесс возвращается в цикл опроса и ожидает поступления других RPC-запросов.

Порожденный процесс вызывает API *alarm* для настройки сигнала SIGALRM, который должен посылаться ему по истечении установленного клиентом интервала времени. Он также вызывает API *signal*, который перехватывает сигнал SIGALRM. Наконец, порожденный процесс вызывает API *pause*, который приостанавливает выполнение этого процесса до получения сигнала SIGALRM.

Когда в порожденный процесс доставляется сигнал SIGALRM, вызывается функция *call_client*. Эта функция настраивает объект *RPC_cls* на работу с функцией обратного вызова клиента и посылает клиентской RPC-функции в качестве аргумента значение, равное времени, которое осталось до сигнала будильника (должно быть равно нулю). После завершения RPC-вызова эта функция активизирует функцию *exit*, которая завершает порожденный процесс.

Программа-клиент в нашем примере называется *aclk_cls.C*:

```
#include <netconfig.h>
#include "aclock.h"
#include "RPC.h"

RPC_svc *svcp = 0;

#define CLNTPROGNUM 0x20000105

/* функция обратного RPC-вызова клиента */
int callback( SVCXPRT* xtrp )
{
    u_long timv;
    /* получить значение времени, оставшегося до сигнала будильника */
    if (svcp->getargs(xtrp, (xdrproc_t)xdr_u_long,
                      (caddr_t)&timv)!=RPC_SUCCESS)
    {
```



```
        cerr << "client: get alarm time fails\n";
        return -1;
    }
    cerr << "client: alarm time left is: " << timv << endl;

    /* возвратить серверу полученные значения */
    if (svcp->reply(xtrp, (xdrproc_t)xdr_void, 0)!=RPC_SUCCESS)
    {
        cerr << "client: send reply failed\n";
        return -2;
    }
    /* выполнять другую работу, затем завершить клиентский процесс */
    exit(0);
}

/* зарегистрировать обратный вызов на RPC-сервере */
int register_callback( char* local_host, char* svc_host, u_long alarm_time)
{
    struct arg_rec argRec;

/* сообщить удаленному серверу хост-имя процесса, номер программы,
   номер версии, номер процедуры и интервал времени до сигнала
   будильника */
    argRec.hostname = local_host;
    argRec.prognum  = svcp->progno();
    argRec.versnum  = CLNTVERNUM;
    argRec.procnum  = CLNTPROCNUM;
    argRec.atime    = alarm_time;

    /* настроить клиентский процесс на соединение с RPC-сервером */
    RPC_cls clnt( svc_host, ACLKPROG, ACLKVER, "netpath");
    if (!clnt.good()) return 1;

    /* вызвать RPC-функцию сервера (set_alarm) */
    if (clnt.call( ACLKPROC, (xdrproc_t)xdr_arg_rec, (caddr_t)&argRec,
                   (xdrproc_t)xdr_void, (caddr_t)0 ) !=RPC_SUCCESS) {
        return 2;
    }
    cerr << "client: " << getpid() <<": RPC called done\n";
    return 0;
}

/* клиентская главная функция */
int main (int argc, char* argv[])
{
    if (argc!=4) {
        cerr << "usage: " << argv[0] << " <local-host> <svc-host>
                                         <transport>\n";
        return 1;
    }
    /* создать серверный процесс для приема обратного вызова
       от удаленного сервера */
       if (!(svcp= new RPC_svc( RPC_ANYFD, argv[3], 0, CLNTVERNUM )))
       return 2;
```

```
    /* определить функцию обратного вызова */
    svcp->add_proc( CLNTPROCNUM, callback );  // определить функцию
                                              // обратного вызова

    /* зарегистрировать обратный вызов на удаленном сервере */
    if (register_callback( argv[1], argv[2], l0)) return 3;

    /* выполнить другую работу ... */

    svcp->run(); /*  ожидать истечения заданного интервала времени
                     до сигнала будильника */
    return 0;
}
```

Клиентский процесс начинает работу созданием объекта *RPC_svc* для регистрации RPC-функции обратного вызова *callback* с помощью демона *rpcbind* (через API *svc_create*). Номера программы, версии и процедуры функции обратного вызова — CLNTPROGNUM, CLNTVERNUM и CLNTPROCNUM соответственно. После создания объекта *RPC_svc* клиент вызывает функцию *register_callback*, для того чтобы сообщить серверу заданный временной интервал и данные обратного вызова. После выполнения функции *register_callback* клиент переходит к выполнению другой работы. Затем он активизирует функцию *RPC_svc::run* и ожидает обратного вызова.

Функция *register_callback* создает объект *RPC_cls* для соединения с RPC-функцией сервера будильника. Она вызывает RPC-функцию сервера и передает следующую информацию: хост-имя машины клиента; номера программы, версии и процедуры функции обратного вызова; заданный интервал времени. Эта информация нужна серверу для вызова клиентской RPC-функции по истечении заданного временного интервала.

Клиентская RPC-функция *callback* вызывается сервером будильника по истечении указанного времени. Функция *callback* вызывает *RPC_svc::getargs*, чтобы извлечь аргумент сервера (время, оставшееся до сигнала будильника). Затем она возвращает серверу полученные значения (через *RPC_svc::reply*). Наконец, функция вызывает *exit* и завершает таким образом клиентский процесс.

Функции XDR-преобразования этой программы содержатся в *ack_xdr.c*:

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "aclock.h"

bool_t
xdr_name_t(register XDR *xdrs, name_t *objp)
{

    register long *buf;

    if (!xdr_string(xdrs, objp, MAXNLEN)) return (FALSE);
```



```
    return (TRUE);
}

bool_t
xdr_arg_rec(register XDR *xdrs, arg_rec *objp)
{

    register long *buf;

    if (!xdr_name_t(xdrs, &objp->hostname)) return (FALSE);
    if (!xdr_u_long(xdrs, &objp->prognum)) return (FALSE);
    if (!xdr_u_long(xdrs, &objp->versnum)) return (FALSE);
    if (!xdr_u_long(xdrs, &objp->procnum)) return (FALSE);
    if (!xdr_u_long(xdrs, &objp->atime)) return (FALSE);
    return (TRUE);
}
```

Эти XDR-функции преобразуют информацию для обратного вызова клиента, указанную в записи *struct arg_rec*, которая посылается серверу в RPC-вызове *set_time*.

Ниже показано, как компилируются и запускаются рассмотренные программы. Предполагается, что серверный процесс работает на машине *saturn*, а клиентский — на машине *fruit*.

На машине *saturn*:

```
saturn%  CC -DSYSV4 -o aclk_svc aclk_svc.C RPC.C aclk_xdr.c \
         -lsocket -lnsl
saturn%  aclk_svc &
```

На машине *fruit*:

```
fruit%  CC -DSYSV4 -o aclk_cls aclk_cls.C RPC.C aclk_xdr.c \
        -lsocket -lnsl
fruit%  aclk_cls fruit saturn netpath
```

12.10. Временный номер RPC-программы

В приведенном выше примере у клиентской RPC-функции заранее определены номера программы, версии и процедуры, что не позволяет запускать в сети одновременно несколько клиентских процессов. Эту проблему можно преодолеть, создав разные версии клиентской RPC-программы, каждая из которых имеет свой номер. Однако в этом случае значительно усложняется сопровождение программ. Более эффективное решение — сгенерировать для каждого клиентского процесса на время его выполнения номер RPC-программы (временный номер). Тогда в локальной сети одновременно сможет работать множество клиентских процессов (сервер различит их по уникальным номерам RPC-программ). Однако это касается версии RPC, реализованной в ОС UNIX System V.4. Метод генерации временных номеров портов поддерживают не все UNIX-системы.

Номера RPC-программ с 0x40000000 по 0x5FFFFFFF зарезервированы для использования в качестве временных номеров. Это позволяет любому

процессу динамически связываться с демоном *rpcbind* и резервировать в качестве номера (или номеров) RPC-программ одно или несколько этих значений при условии, что номер не задействован другими процессами. Временные номера RPC-программ, затребованные процессом, освобождаются после его завершения и могут быть использованы другими процессами.

Временный номер RPC-программы освобождается с помощью статической функции *RPC_svc::gen_progNum*. Эта функция вызывает API *rpcb_set* для каждого номера из диапазона 0x40000000 — 0x5FFFFFFF и спрашивает у демона *rpcbind*, присвоен ли тот или иной номер какому-либо процессу. Функция находит младший из свободных временных номеров, и *rpcb_set* регистрирует этот номер и указанный номер версии в *rpcbind*.

API *rpcb_set* имеет следующий прототип:

```
#include <rpc/rpc.h>

boot_t rpcb_set (const u_long prognum, const u_long versnum,
                 const struct netconfig* netconf, const struct netbuf* addr);
```

Аргументы *prognum* и *versnum* — это затребованные номера RPC-программы и версии, которые должны быть присвоены вызывающему процессу. Аргумент *netconf* содержит информацию о способе передачи данных, используемом вызывающим процессом. Аргумент *addr* — это сетевой адрес вызывающего процесса.

При успешном выполнении эта функция возвращает значение TRUE, и затребованные номера программы и версии регистрируются в *rpcbind* для процесса с указанным адресом и транспортным протоколом. В случае неудачи функция возвращает значение FALSE.

Аргументы *netconf* и *addr* функции *RPC_svc::gen_progNum* получить не так просто, особенно если обработчик сервера создан посредством API *svc_create*. Если же для этого был использован API *svc_tli_create*, то доступ к значениям *netconf* и *addr* сервера осуществить легко.

Чтобы использовать переходные номера программ, в класс *RPC_cls* можно добавить следующую перегруженную функцию-конструктор *RPC_svc::RPC_svc*:

```
RPC_svc( int fd, char* transport, u_long progno, u_long versno)
{
    rc = 0;          /* статус неудачного завершения */

    struct netconfig* nconf = getnetconfigent(transport);
    if (!nconf) {
        cerr << "invalid transport:" << transport << endl;
        return;
    }

    /* создать обработчик сервера */
    SVCXPRT * xprt = svc_tli_create(fd, nconf, 0, 0, 0);
```



```
    if (!xprt) {
        cerr << "create server handle fails\n";
        return;
    }

    if (!progno) {          /* сгенерировать переходный номер */
        progno = gen_progNum(versno, nconf, &xprt->xp_ltaddr);
        nconf = 0;          /* дать svc_reg указание не общаться
                               с rpcbind */
    }

    if (svc_reg(xprt, progno, versno, dispath, nconf)==FALSE)
        cerr << "register prognum failed\n";
    else {
        prognum = progno; vernum = verno;
        rc = 1;
    }
    freenetconfigent( nconf );
}
```

В этой перегруженной функции-конструкторе вызывается функция *getnetconfigent*, которая возвращает указатель на запись данных типа *struct netconf* с информацией о способе передачи данных по сети для данного аргумента функции. После того как этот указатель получен (он содержится в переменной *nconf*), вызывается API *svc_tli_create*, который создает обработчик сервера и возвращает указатель на него. Переменная *netconf* освобождается с помощью функции *freenetconfigent*.

Функция *svc_tli_create* имеет следующий прототип:

```
#include <rpc/rpc.h>

SVCXPR* svc_tli_create (const int fd, const struct netconfig* netconf,
        const struct t_bind* baddr, const u_int sendsz, const u_int recsz);
```

Аргумент *fd* — это дескриптор файла, обозначающий файл устройства, связанный с выбранным механизмом транспортировки. Если его значение задано как RPC_ANYFD, то используется файл устройства, определенный аргументом *netconf*. Адрес, присвоенный серверу, определяется аргументом *baddr*, если его значение не NULL; в противном случае используется адрес, принятый по умолчанию для данного транспортного протокола. Аргументы *sendsz* и *recvsz* задают необходимые размеры буферов приема и передачи для обработчика сервера. Если их значения равны нулю, то используются стандартные для транспортного протокола размеры.

В случае неудачи API *svc_tli_create* возвращает NULL; в противном случае возвращается указатель на обработчик сервера.

Если функция *svc_tli_create* выполняется успешно и заданное значение аргумента *progno* равно нулю, вызывается функция *RPC_svc::gen_progNum*,

которая генерирует временный номер программы. После этого активизируется API *svc_reg*, который связывает номер программы и номер версии с соответствующей функцией-диспетчером.

Функция API *svc_reg* имеет следующий прототип:

```
#include <rpc/rpc.h>

int svc_reg (const SVCXPR* xprt, const u_long prognum,
                    const u_long versnum, const void (*dispath)(...),
                    const struct netconfig* netconf);
```

Аргумент *xprt* — это обработчик сервера. Аргументы *prognum* и *versnum* — это номер программы и номер версии, связанные с функцией-диспетчером, заданной аргументом *dispatch*. Аргумент *netconf* задает метод транспортировки, который может использоваться для регистрации RPC-функции и функции-диспетчера демоном *rpcbind*. Если аргумент *netconf* имеет значение NULL, такая регистрация не требуется.

В функции-конструкторе *RPC_svc* функция *svc_reg* вызывается с нулевым значением аргумента *netconf* (если вызывается *RPC_svc::gen_progNum*). Это объясняется тем, что функция *RPC_svc::gen_progNum* автоматически регистрирует RPC-функцию демоном *rpcbind*.

В случае неудачи API *svc_reg* возвращает FALSE; в противном случае возвращается TRUE и RPC-сервер настраивается нормально.

Программу *aclk_cls.C* можно переписать с использованием перегруженной функции-конструктора *RPC_svc::RPC_svc*, которая создает временный номер программы. Новая программа *aclk_cls2.C* выглядит следующим образом:

```
#include <netconfig.h>
#include "aclock.h"
#include "RPC.h"

RPC_svc *svcp = 0;

#define CLNTPROGNUM 0x20000105

/* функция обратного RPC-вызова клиента */
int callback( SVCXPRT* xtrp )
{
    u_long timv;
    /* получить значение времени, оставшегося до сигнала будильника */
    if (svcp->getargs(xtrp, (xdrproc_t)xdr_u_long,
                      (caddr_t)&timv)!=RPC_SUCCESS)
    {
        cerr << "client: get alarm time fails\n";
        return -1;
    }
    cerr << "client: alarm time left is: " << timv << endl;
```



```
    /* возвращает серверу полученные значения */
    if (svcp->reply(xtrp, (xdrproc_t)xdr_void, 0)!=RPC_SUCCESS)
    {
        cerr << "client: send reply failed\n";
        return -2;
    }
    /* выполнять другую работу, затем завершить клиентский процесс */
    exit(0);
}

/* зарегистрировать обратный вызов на RPC-сервере */
int register_callback( char* local_host, char* svc_host, u_long alarm_time)
{
    struct arg_rec argRec;

    /* сообщить удаленному серверу хост-имя процесса, номер
       программы, номер версии, номер функции и заданный интервал
       времени до сигнала будильника */
    argRec.hostname = local_host;
    argRec.prognum  = svcp->progno();
    argRec.versnum  = CLNTVERNUM;
    argRec.procnum  = CLNTPROCNUM;
    argRec.atime    = alarm_time;

    /* настроить клиентский объект на соединение с RPC-сервером */
    RPC_cls clnt( svc_host, ACLKPROG, ACLKVER, "netpath");
    if (!clnt.good()) return 1;

    /* вызвать RPC-функцию сервера (set_alarm) */
    if (clnt.call( ACLKPROC, (xdrproc_t)xdr_arg_rec, (caddr_t)&argRec,
                   (xdrproc_t)xdr_void, (caddr_t)0 ) !=RPC_SUCCESS)
{
        return 2;
    }
    cerr << "client: " << getpid() <<": RPC called done\n";
    return 0;
}

/* главная клиентская функция */
int main (int argc, char* argv[])
{
    if (argc!=4) {
        cerr << "usage: " << argv[0] << " <local-host> <svc-host>
                                        <transport>\n";
        return 1;
    }
    /* создать серверный объект для приема обратного вызова от
       удаленного сервера */
    if (!(svcp= new RPC_svc( RPC_ANYFD, argv[3], 0, CLNTVERNUM )))
        return 2;
```

```
    /* определить функцию обратного вызова */
    svcp->add_proc( CLNTPROCNUM, callback );

    /* зарегистрировать обратный вызов на удаленном сервере */
    if (register_callback( argv[1], argv[2], 10)) return 3;

    /* выполнить другую работу ... */

    svcp->run(); /* ожидать истечения заданного интервала времени
                    до сигнала будильника */
    return 0;
}
```

Новая программа отличается от описанной в разделе 12.9 одной строкой, в которой с помощью оператора *new* в главной функции создается указатель *RPC_svc*.

Эту программу можно компилировать и запускать так, как показано в разделе 12.9. Результаты выполнения старой и новой программ одинаковы.

12.11. RPC-услуги на базе inetd

RPC-серверы — это, как правило, процессы-демоны, которые работают непрерывно, ожидая поступления RPC-вызовов от клиентов. Недостаток такой схемы в том, что системные ресурсы, выделенные этим процессам (например, элементы таблицы процессов), не могут использоваться другими процессами даже тогда, когда эти демоны простаивают. Повысить эффективность использования системных ресурсов можно с помощью программ мониторинга портов, например *inetd*, которые позволяют управлять сетевыми адресами для RPC-услуг тогда, когда RPC-серверы не выполняются. Когда поступает RPC-запрос, монитор порта порождает RPC-сервер, который отвечает на этот запрос и самозавершается после предоставления услуги. Таким образом, системные ресурсы выделяются RPC-серверу только на то время, которое необходимо для ответа на запрос клиента.

В большинстве коммерческих UNIX-систем для мониторинга порта используется программа *inetd*. Она запускается при начальной загрузке системы и из файла */etc/inetd.conf* получает сетевые адреса, которыми может управлять. Каждый элемент этого файла имеет следующий синтаксис:

```
<сервис> <средство_транспортировки> <протокол> <wait> <uid>
        <программа> <арг>
```

Такая запись означает, что если поступает запрос на указанный сервис (<сервис>), *inetd* должен выполнить программу <программа>, указав значение <арг> в качестве ее аргумента. Действующий UID выполняемой программы задается полем <uid>, а для связи с клиентским процессом используются указанные средство транспортировки и протокол. В полях <средство_транспортировки> и <протокол> используются следующие значения:

Средство транспортировки	Протокол
stream	tcp
dgram	udp

Для сервисов, использующих гнезда, в поле <wait> указываются следующие значения: *nowait* — для транспортировки с установлением соединения (tcp), *wait* — для транспортировки без установления соединения (udp). Для сервисов на базе TLI в поле <wait> обычно устанавливается *wait*.

Адрес порта сервиса указывается в файле */etc/services* следующим образом:

```
<сервис>        <порт>/<протокол>
```

Предположим, что в файле */etc/inetd.conf* имеется следующая запись:

```
login stream tcp nowait root /etc/in.rlogind in.rlogind
```

Когда удаленный пользователь попытается зарегистрироваться на локальной хост-машине, программа *inetd* должна выполнить программу */etc/in.logind* от имени пользователя *root*. Процесс *rlogin* будет использовать транспортный протокол TCP/IP и адрес порта 513, как указано в файле */etc/services*:

```
login    513/tcp
```

Чтобы программа *inetd* проверяла наличие конкретного RPC-запроса, в файл */etc/inetd.conf* следует ввести запись для этого RPC-сервера:

```
<номер_прог>/<номер_верс> <средство_транспортировки> <протокол> <wait> \
<uid> <программа> <арг>
```

Здесь <номер_прог> и <номер_верс> — это номера программы и версии RPC-сервера. Остальные поля такие же, как для вышеупомянутого RPC на базе ONC. В ОС UNIX System V.4 поле <средство_транспортировки> может содержать значение *tli*, если обработчик RPC-сервера создается на основе TLI, а поле <протокол> может содержать значение *rpc/tcp*, *rpc/udp* или *rpc/**. Значение *rpc/** указывает на то, что сервер может пользоваться любым механизмом транспортировки, который поддерживается интерфейсом TLI.

Например, в программе вывода содержимого каталога из раздела 12.7.4 используется номер программы 0x200100 и номер версии 1. Чтобы *inetd* поддерживал этот сервис, необходимо ввести в файл */etc/inetd.conf* следующую запись (подразумевается, что выполняемый файл RPC-сервера имеет имя */proj/scan_svc3*):

Для ONC:

```
# 536871168 is same as 0x20000100
536871168/1  stream  tcp  wait  root  /proj/scan_svc3  scan_svc3
```

Для UNIX System V.4:

```
536871168/1  tli  rpc/*  wait  root  /proj/scan_svc3  scan_svc3
```

Помимо конфигурирования *inetd*, RPC-сервер должен создать свой обработчик *RPC_svc* (класс *RPC_svc* описан в разделе 12.5) с помощью конструктора:

```
RPC_svc::RPC_svc(int fd, char* transport, unsigned long progno,
                 unsigned long versnum);
```

Для того чтобы аргумент *fd* конструктора *RPC_svc* соответствовал входящему RPC-запросу, демон *inetd* присваивает ему значение 0. В приведенном выше примере сервер должен создать свой обработчик *RPC_svc* следующим образом:

```
RPC_svc *svcp = new RPC_svc(0, "tcp", 0x20000100, 1);
```

Ниже перечислены API, с помощью которых конструктор *RPC_svc::RPC_svc* создает обработчик *RPC_svc*.

В ONC:

- для создания обработчика сервера вызывается *svctcp_create* (для механизма транспортировки, использующего потоки) или *svcudp_create* (для механизма транспортировки на основе дейтаграмм);
- для регистрации функции-диспетчера, которая будет вызываться при поступлении RPC-вызова, вызывается *svc_reg*.

В UNIX System V.4:

- чтобы создать объект типа *struct netconfig* для необходимого транспортного протокола (*tcp* или *udp*), вызывается *getnetconfigent*;
- для создания обработчика сервера вызывается *svc_tli_create*;
- для регистрации функции-диспетчера, которая будет вызываться при поступлении RPC-вызова, вызывается *svc_reg*.

Тот, кого интересует последовательность вызова этих API, может обратиться к коду конструктора *RPC_svc*, приведенному в разделе 12.5.

Мы перечислили все изменения, которые необходимо внести, чтобы получить возможность управлять запросами RPC-сервиса с помощью *inetd*. Остальная часть кода сервера не изменилась. Более того, в XDR-функциях и клиентских программах, выполняющих RPC-вызовы, вообще ничего изменять не нужно.

В качестве последнего примера возьмем программу вывода списка файлов каталога, приведенную в разделе 12.7.4, и перепишем ее так, чтобы сервер использовал *inetd* для управления RPC-запросами. Тексты программы-клиента (*scan_cls2.C*) и XDR-функции (*scan_xdr.c*) содержатся в разделе 12.7.4, поэтому здесь не повторяются. Ниже приведена модифицированная серверная программа *scan_svc3.C*.

```
/* серверная программа: использование API RPC низкого уровня */
/* вызов: scan_svc3  <ttl> */

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



```
#include <dirent.h>
#include <string.h>
#include <malloc.h>
#include <sys/stat.h>
#include <sys/resource.h>
#include <signal.h>
#include "scan2.h"
#include "RPC.h"

char* transp = "tcp";

extern int errno;

static RPC_svc *svcp = 0;
static int work_in_progress = 0;
static int ttl = 60;  /* время жизни 60 секунд */

/* RPC-функция */
int scandir( SVCXPRT* xtrp )
 {
    DIR *dirp;
    struct dirent *d;
    infolist  nl, *nlp;
    struct stat statv;
    res res;
    argPtr darg = 0;
    work_in_progress = 1; /* процесс не уничтожен сигналом
                             будильника */

    /* получить аргумент функции от клиента */
    if (svcp->getargs( xtrp, (xdrproc_t)xdr_argPtr,
                       (caddr_t)&darg)!=RPC_SUCCESS)
        return -1;

    cerr << "server: get dir: '" << darg->dir_name << "', lflag="
                                << darg->lflag << endl;

    /* начать просмотр затребованного каталога */
    if (!(dirp = opendir(darg->dir_name))) {
        res.errno = errno;
        (void)svcp->reply(xtrp, (xdrproc_t)xdr_res, (caddr_t)&res);
        return -2;
    }
    /* освободить память, выделенную в предыдущем RPC-вызове */
    xdr_free((xdrproc_t)xdr_res, (char*)&res);
    /* занести информацию о файлах в res как возвращаемые значения */
    nlp = &res.res_u.list;
    while (d=readdir(dirp)) {
        nl = *nlp = (infolist)malloc(sizeof(struct dirinfo));
        nl->name = strdup(d->d_name);
        nlp = &nl->next;
        if (darg->lflag) {
```

```
            char pathnm[256];
            sprintf(pathnm,"%s/%s",darg->dir_name,d->d_name);
            if (!stat(pathnm,&statv)) {
                nl->uid = statv.st_uid;
                nl->modtime = statv.st_mtime;
            }
        }
    }
    *nlp = 0;
    res.errno = 0;
    closedir(dirp);
    /* передать список файлов каталога клиенту */
    if (svcp->reply(xtrp, (xdrproc_t)xdr_res, (caddr_t]&res]!=RPC_SUCCESS)
        return -2;

    work_in_progress = 0;   /* процесс может быть уничтожен сигналом
                               будильника */

    return RPC_SUCCESS;
}

/* программа обработки сигналов */
void done( int sig )
(
    if (work_in_progress] (
        /* re-schedule time-out */
        signal( SIGALRM, (void(*)(int]) done );
        alarm( ttl );
    }
    char msg[ 1024 ];
    time_t tim = time(0);
    sprintf( msg, "svc (%d] exiting: %s", getpid(], ctime(&tim) ];
    write( 1, msg, strlen(msg] );
    exit(0);
 }

int main(int argc, char* argv(])
[
    char msg[ 1024 ];

    FILE *fp = fopen("/dev/console","w");
    fprintf( fp, "sv3: argc=%d, argv(1]=%s\n", argc, argc > 1 ?
                                                argv(1] : "Nil" );

    if (argc > 1 && sscanf(argv(1],"%d",&ttl]!=1) (
        cerr << "Invalid argument: " << argv(1) << endl;
        return 1;
    }

    switch (fork()) {
        case  0:  break;
        case -1:  perror( "fork");
```



```
        default:  return errno;
    }

    /* закрыть все потоки ввода-вывода, кроме дескриптора 0 */
    struct rlimit rls;
    rls.rlim_max = 0;
    getrlimit(RLIMIT_NOFILE, &rls);
    if (rls.rlim_max == 0)  {
        fprintf( fp, "getrlimit failed\n");
        return 1;
    }
    for (int i = 1; i < rls.rlim_max; i++)
        (void) close(i);

    /* все выходные сообщения переадресованы на системную консоль */
    int fd = open("/dev/console", 2);
    (void) dup2(fd, 1);
    (void) dup2(fd, 2);

    setsid();

    time_t tim = time(0);
    sprintf( msg, "svc (%d)starting: %s", getpid(), ctime(&tim) );
    write( 1, msg, strlen(msg) );

    /* теперь создать обработчик RPC-сервера */
    svcp = new RPC_svc( 0, transp, SCANPROG, SCANVER );
    if (!svcp || !svcp->good()) {
        sprintf(msg, "RPC_svc failed\n");
        write(1, msg, strlen(msg) );
        exit( 1 );
    }

    svcp->add_proc( SCANDIR, scandir );

    /* завершить демон после 60 секунд активности */
    signal( SIGALRM, (void(*)(int)) done );

    svcp->run();

    return 0;
}
```

В этой программе сервер создает обработчик *RPC_svc* для дескриптора файла 0. При этом используется механизм транспортировки *tcp*. Он регистрирует функцию *scandir* как такую, которая может быть вызвана клиентами, а затем устанавливает сигнал SIGALRM для передачи самому себе. Последняя операция обусловлена тем, что *inetd* не порождает новый процесс для запроса сервиса, если уже есть работающий серверный процесс (это делается для того, чтобы не создавать лишние процессы). Таким образом, сервер, порожденный демоном *inetd*, после обслуживания запроса чаще всего

блокируется на определенное время для того, чтобы он мог перехватить следующий запрос на обслуживание.

После регистрации функции *done* как обработчика сигнала SIGALRM программа-сервер вызывает *RPC_svc::run*. Поскольку уже имеется ожидающий запрос, немедленно вызывается функция *scandir*. После того как функция *scandir* возвращает управление, сервер блокируется в функции *RPC_svc::run*, ожидая следующего RPC-вызова. Сервер "живет" максимум 60 секунд, в течение которых должен поступить новый RPC-вызов. Если это происходит, функция *done* изменяет установку будильника так, чтобы процесс мог работать еще 60 секунд. При каждом вызове функции *scandir* устанавливается глобальная переменная *work_in_progress*, которая сбрасывается после выполнения функции *scandir*. В зависимости от значения этой переменной функция *done* либо завершает процесс, либо перезапускает будильник.

Клиентская и серверная программы компилируются в системе UNIX System V.4 следующим образом (в ONC-системах они компилируются с опцией -DSYSV4):

```
% CC -c scan2_xdr.c RPC.C
% CC -DSYSV4 -o scan_svc3 scan_svc3.C scan2_xdr.o \
          RPC.o -lsocket -lnsl
% CC -DSYSV4 -o scan_cls2 scan_cls2.C scan2_xdr.o \
          RPC.o -lsocket -lnsl
```

Элемент файла */etc/inetd.conf* для сервера *scan_svc3* в формате System V.4 выглядит следующим образом:

```
536871168/1  tli  rpc/*  wait  root  /proj/scan_svc3  scan_svc4
```

Ниже показан пример запуска клиентской и серверной программ. Серверная часть находится на машине *fruit*, а клиентскую можно запускать на любой машине, соединенной с *fruit*. Вручную запускается только программа-клиент, а серверная программа выполняется демоном *inetd*:

```
% scan_cls2 fruit
....
....
...scan_cls2.C
...scan_svc2.C
...RPC.C
...RPC.h
...scan2_xdr.c
...scan2.h
...scan_svc2
...scan_cls2
Prog 536871168 (version 1) is alive
```


12.12. Заключение

В этой главе рассмотрены три метода создания RPC-программ: с использованием системных библиотечных RPC-функций; с помощью компилятора *rpcgen*, создающего специализированные RPC-функции и клиентские главные программы; с использованием классов RPC, которые позволяют создавать полные специализированные клиентские и серверные RPC-программы.

Наиболее гибкий среди них, пожалуй, последний, потому что пользователи могут полностью контролировать содержимое клиентских и серверных программ, а также управлять механизмами транспортировки, которые используются их приложениями. Кроме того, классы RPC "скрывают" большинство низкоуровневых API RPC, поэтому трудоемкость программирования с использованием этих классов не намного превышает трудоемкость программирования с помощью компилятора *rpcgen*.

В главе представлены примеры применения различных методов программирования RPC, рассмотрены широковещательные RPC, асинхронные обратные вызовы (от клиентов к серверам), аутентификация, генерация временных номеров RPC-программ, управление RPC-запросами с помощью *inetd*. Читатели могут использовать эти примеры как отправную точку и модифицировать их для создания собственных RPC-приложений.

ГЛАВА 13

Многопотоковое программирование

Поток выполнения — это элемент кода программы, выполняемый последовательно. Большинство UNIX-приложений — однопотоковые программы, так как каждая из них выполняет в каждый момент времени только один элемент кода. Например, однопотоковый процесс получает команду от пользователя, выполняет ее, сообщает пользователю результаты, а затем ожидает следующую команду. Пользователь должен дождаться, пока процесс закончит выполнение команды, и лишь затем вводить следующие команды.

В многопотоковой программе в каждый момент времени могут выполняться "параллельно" несколько элементов кода, при этом каждый элемент кода выполняется одним потоком управления. Работая с многопотоковым процессом, пользователь может вводить команды непрерывно, одну за другой, и процесс выполняет все команды параллельно.

Многопотоковое программирование можно использовать для разработки приложений, которые могут выполняться параллельно. Эти приложения можно запускать на любых многопроцессорных системах, эффективно используя аппаратные ресурсы. В частности, если многопотоковое приложение запускается на системе с M процессорами, то все его потоки могут выполняться одновременно, каждый — отдельном процессором. Следовательно, производительность такого приложения можно увеличить в N раз, где N — максимальное число свободных в данный момент процессоров (N меньше или равно M).

Производительность многопотокового приложения можно улучшить даже в том случае, если оно запускается на однопроцессорной системе. Например, если один из потоков приложения блокируется каким-то системным вызовом, на этом процессоре может выполняться другой поток. Таким образом сокращается общее время выполнения приложения.

Помимо этих преимуществ, многопотоковое программирование — хорошее дополнение к объектно-ориентированному программированию. Это обусловлено тем, что каждое объектно-ориентированное приложение состоит из набора объектов, взаимодействующих друг с другом для выполнения задач. Каждый из этих объектов — независимый компонент, который может выполняться потоком и запускаться параллельно с другими объектами, что позволяет значительно повысить производительность таких приложений. Например, в многопотоковом объектно-ориентированном оконном интерфейсе каждое меню, кнопка, текстовое поле и прокручиваемое окно может обрабатываться потоком. Следовательно, любое количество этих оконных объектов можно активизировать друг за другом, не ожидая завершения обработки других объектов. Таким образом, GUI-приложение в целом легче реагирует на действия пользователя, "интерактивнее", чем его однопотоковый вариант.

Потоки выполнения отличаются от порожденных процессов, создаваемых функцией API *fork*:

- Потоками выполнения можно управлять либо с помощью библиотечных функций пользовательского уровня, либо с помощью ядра операционной системы. Процессами, которые порождаются системным вызовом *fork*, управляет ядро операционной системы. Вообще говоря, потоки более эффективны и требуют гораздо меньше внимания со стороны ядра в процессе создания и управления.
- Все потоки выполнения в процессе совместно используют сегменты данных и кода. Порожденный процесс имеет собственную копию виртуального адресного пространства, отдельную от родительского процесса. Таким образом, потоки используют гораздо меньше системных ресурсов, чем порожденные процессы.
- Функции *exit* или *exec*, вызываемые потоком, завершают все потоки в этом процессе. Если же эти функции вызывает порожденный процесс, то на родительский процесс ее действие не распространяется.
- Если поток модифицирует в процессе какую-то глобальную переменную, эти изменения видимы для остальных потоков этого процесса. Поэтому для потоков, обращающихся к совместно используемым данным, необходима синхронизация. Во взаимоотношениях между порожденным и родительским процессами эта проблема не возникает.

Теперь перечислим преимущества многопотокового программирования:

- повышается производительность процессов и ускоряется реакция на действия пользователя;
- процесс может использовать все свободные аппаратные средства многопроцессорной системы, в которой выполняется многопотоковое приложение;
- программистам могут структурировать код в независимо выполняемые компоненты;
- снижается необходимость использования функции *fork* для создания порожденных процессов и таким образом увеличивается производительность каждого процесса (реже выполняется переключение контекста); в управлении выполнением потоков в меньшей степени участвует ядро системы;
- многопотоковое программирование — оптимальный способ повышения производительности объектно-ориентированных приложений, рассчитанных на использование в многопроцессорных системах.

Недостаток многопотокового программирования состоит в том, что пользователи должны обеспечить синхронизацию потоков в каждой программе. Синхронизация нужна для того, чтобы потоки не делали случайных ошибок при чтении и записи совместно используемых данных и не могли уничтожить свой процесс системным вызовом *exit* или *exec*.

Разработка технологии многопотокового программирования началась в середине 80-х годов. Поставщики разных вариантов ОС UNIX предлагали различные версии интерфейсов многопотокового программирования. Комитет POSIX разработал комплект многопотоковых API, который вошел в стандарт POSIX.1c. В этой главе будут рассмотрены и многопотоковые API ОС Solaris 2.x (разработка фирмы Sun Microsystems), и многопотоковые API стандарта POSIX.1c, но особое внимание мы уделим API фирмы Sun. Причина этого в том, что стандарт POSIX.1c — новый и его пока поддерживает не очень много поставщиков UNIX, а вот многопотоковые API фирмы Sun доступны прикладным программистам уже довольно давно. К тому же многопотоковые API Sun имеют близкое сходство с многопотоковыми API POSIX.1c. Благодаря этому приложения, построенные на API Sun, можно свободно конвертировать в стандарт POSIX.1c.

13.1. Структура и методика использования потоков выполнения

Поток выполнения состоит из следующих элементов:

- идентификатора потока;
- динамического стека;
- набора регистров (счетчик команд, указатель стека);
- сигнальной маски;
- значения приоритета;
- специальной памяти.

Поток выполнения создается функцией *thr_create* (в POSIX.1c — *pthread_create*). Каждому потоку присваивается идентификатор, уникальный среди всех потоков процесса. Вновь созданный поток наследует сигнальную маску процесса, динамический стек, значение приоритета и набор регистров. Динамический стек и регистры (счетчик команд и указатель стека) позволяют потоку выполняться независимо от других потоков. Поток может изменить унаследованную сигнальную маску и выделить динамическую память для хранения своих данных.

Потоку при создании назначается соответствующая функция. Поток завершается, когда эта назначенная функция возвращает результат или когда поток вызывает функцию *thr_exit* (в POSIX.1c — *pthread_exit*). Когда в процессе создается первый поток, то фактически создаются два потока: один — для выполнения указанной функции, а другой — для продолжения выполнения процесса. Последний поток завершается, когда функция *main* возвращает результат или он сам вызывает функцию *thr_exit*.

Все потоки в процессе совместно используют одни сегменты данных и кода. Когда один поток записывает данные в глобальные переменные в процессе, остальные потоки сразу же видят эти изменения. Если какой-либо поток вызывает API *exit* или API *exec*, то завершаются все потоки и сам процесс. Поэтому завершающийся поток, если он не хочет разрушить процесс, в котором выполняется, должен вызывать функцию *thr_exit*.

Поток может изменить свою сигнальную маску с помощью функции *thr_sigsetmask* (в POSIX.1c — *pthread_sigmask*). Сигнал, передаваемый процессу, получат все потоки, которые не замаскировали его. Поток может посылать сигналы в другие потоки этого же процесса, используя функцию *thr_kill* (в POSIX.1c — *pthread_kill*), но не может посылать сигналы в потоки другого процесса, так как уникальность идентификаторов потоков не распространяется на другие процессы. Для настройки собственного механизма обработки сигналов поток может использовать API *signal* или *sigaction*.

Потоку присваивается целочисленное значение приоритета. Чем больше это значение, тем чаще планируемое выполнение потока. Значение приоритета потока можно запросить с помощью функции *thr_getprio* и изменить с помощью функции *thr_setprio* (в POSIX.1c — *pthread_attr_getschedparam* и *pthread_attr_setschedparam* соответственно). Поток может намеренно передать выполнение другим потокам с таким же приоритетом; для этого используется функция *thr_yield* (в POSIX.1c — *sched_yield*). Кроме того, поток может ожидать завершения другого потока и получить его код возврата с помощью функции *thr_join* (в POSIX.1c — *pthread_join*).

В API фирмы Sun поток может, пользуясь функциями *thr_suspend* и *thr_continue*, приостанавливать и возобновлять выполнение другого потока. Если какая-то функция выполняется множеством потоков и содержит статические или глобальные переменные, которые используются разными потоками, нужно создать специальную память для хранения этих фактических данных по каждому потоку. Эта специальная память выделяется с помощью функций *thr_keycreate*, *thr_setspecific* и *thr_getspecific*.

13.2. Потоки и облегченные процессы

Многопотоковые библиотечные функции, разработанные фирмой Sun Microsystems, создают облегченные процессы (lightweight processes, LWP), выполнение которых планируется ядром. Такие процессы похожи на виртуальные процессоры тем, что многопотоковые библиотечные функции управляют выполнением потоков в процессе, связывая их с LWP. Если связанный с LWP поток приостанавливается (например, функцией *thr_yield* или *thr_suspend*), этот LWP можно связать с другим потоком, функцию которого он

должен будет выполнять. Если LWP выполняет системный вызов от имени потока, он остается связанным с этим потоком до возврата из вызова. Если все LWP, связанные с потоками, заблокированы системными вызовами, многопотоковые библиотечные функции создают новые LWP, с которыми могут быть связаны потоки, ожидающие выполнения. Таким образом обеспечивается непрерывность выполнения процесса. Наконец, если LWP больше, чем потоков в процессе, то в целях экономии системных ресурсов многопотоковые библиотечные функции удаляют лишние LWP.

Большинство потоков не связаны, поэтому их можно связать с любыми свободными LWP. Вместе с тем процесс может создать один и более потоков, постоянно связанных с облегченными процессами. Эти потоки называются *связанными потоками*. Такое связывание потоков используется главным образом в тех случаях, когда потоки должны:

- планироваться ядром для обработки в режиме реального времени;
- иметь собственные альтернативные сигнальные стеки;
- иметь собственные будильники и таймер.

Взаимосвязь потоков, облегченных процессов и аппаратных процессоров показана на рис. 13.1.

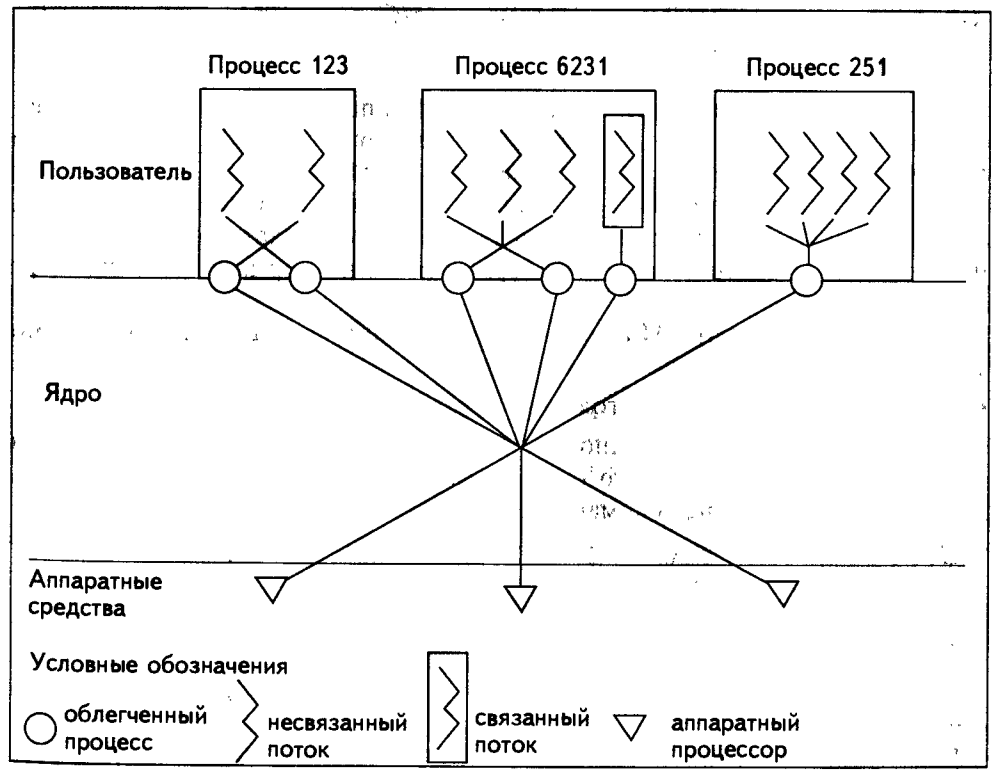


Рис. 13.1. Взаимосвязи при планировании потоков, облегченных процессов и аппаратных процессоров

На рис. 13.1 в процессе 123 есть два несвязанных потока, которые планируются на два LWP. В процессе 6231 — три несвязанных потока, которые планируются на два LWP, и один связанный, который выполняется третьим LWP. В процессе 251 — четыре несвязанных потока, которые планируются на один LWP. Несвязанные потоки в каждом процессе планируются многопотоковыми библиотечными функциями к связыванию с LWP и выполнению в этом процессе. LWP всех процессов, в свою очередь, планируются ядром к выполнению на трех имеющихся аппаратных процессорах.

В POSIX.1c у потоков есть атрибут, который называется *областью действия конкуренции при планировании* (scheduling contention scope). Если этот атрибут установлен в PTHREAD_SCOPE_PROCESS, то данным потоком управляют библиотечные функции пользовательского уровня и он является "несвязанным". Все потоки с этим атрибутом совместно используют ресурсы процессора, доступные для содержащего их процесса. Если же вышеупомянутый атрибут установлен в PTHREAD_SCOPE_SYSTEM, то данным потоком управляет ядро операционной системы и он считается "связанным". В стандарте POSIX.1c не указано, как ядро должно обрабатывать "связанный" поток.

13.3. API потоков выполнения фирмы Sun Microsystems

В этом разделе рассматриваются только API потоков выполнения, разработанные фирмой Sun Microsystems. API потоков выполнения стандарта POSIX.1c будут рассмотрены в разделе 13.4. Мы обсуждаем эти потоки выполнения отдельно, чтобы не запутать читателей. Прочитав раздел 13.4, вы узнаете о соответствии API Sun и POSIX.1c, благодаря которому многопотоковые приложения можно преобразовывать из формата Sun в стандарт POSIX.1c.

Чтобы использовать API потоков выполнения фирмы Sun, необходимо сделать следующее:

- включить в программу заголовок <thread.h>;
- откомпилировать и скомпоновать программу с опцией *-lthread*. Если указывается опция *-lC*, то опцию *-lthread* нужно поместить перед ней. Например, следующая команда компилирует многопотоковую C++-программу *x.C*:

```
% CC x.C -o x -lthread -lC
```

Если не указано иного, то большинство описанных ниже API при успешном завершении возвращают 0, а в случае неудачи — -1. В последнем случае может вызываться функция *perror*, которая выводит сообщения об ошибках.

13.3.1. Функция thr_create

Прототип функции *thr_create* выглядит следующим образом:

```
#include <thread.h>

int thr_create (void* stackp, size_t stack_size, void* (*funcp)(void*),
                void* argp, long flags, thread_t* tid_p);
```

Эта функция создает новый поток для выполнения функции, адрес которой задан аргументом *funcp*. Функция, указанная в этом аргументе, должна принимать один входной аргумент типа *void** и возвращать данные такого же типа. Фактический аргумент, который передается функции *funcp*, когда начинает выполняться новый поток, указывается в аргументе *argp*.

Аргументы *stackp* и *stack_size* задают адрес определяемой пользователем области памяти и ее размер в байтах. Эта память используется в качестве динамического стека нового потока. Если аргумент *stackp* принимает значение NULL, функция выделяет область стека размером *stack_size* байтов. Если значение *stack_size* равно нулю, функция использует стандартное системное значение, а именно — один мегабайт виртуальной памяти. Пользователям очень редко приходится собственноручно выделять область памяти для стека потока. Таким образом, аргументы *stackp* и *stack_size* обычно принимают значения NULL и нуль соответственно.

Аргумент *flags* может иметь нулевое значение. В таком случае новому потоку не нужно присваивать какие-либо специальные атрибуты. Значение аргумента *flags* может состоять из одного или нескольких следующих битовых флагов:

Значение аргумента *flags*	Смысл
THR_DETACHED	Создает отсоединенный поток. Это означает, что когда поток завершается, все его ресурсы и присвоенный ему идентификатор можно использовать повторно для другого потока. Ни один поток не должен ожидать его завершения (через функцию *thr_join*)
THR_SUSPENDED	Приостанавливает выполнение нового потока до тех пор, пока другой поток не вызовет функцию *thr_continue*, которая позволит возобновить его выполнение
THR_BOUND	Создает постоянно связанный поток
THR_NEW_LWP	Создает новый LWP вместе с новым потоком
THR_DAEMON	Создает новый поток-демон. Как правило, многопотоковый процесс завершается, когда завершаются все его потоки. Если же процесс содержит один или несколько потоков-демонов, то он завершается сразу же по завершении всех его потоков, которые не являются демонами

Идентификатор нового потока возвращается через аргумент *tid_p*. Если этому аргументу присвоено фактическое значение NULL, никакой идентификатор не возвращается. Тип данных идентификатора потока — *thread_t*.

Функция *thr_create* может не выполниться в тех случаях, когда недостаточно системной памяти для создания нового потока, когда аргумент *stackp* содержит неверный адрес или когда значение аргумента *stack_size* не равно нулю и меньше установленного системой минимального предела. Установленный системой минимально допустимый размер стека можно узнать с помощью функции *thr_min_stack*:

```
size_t    thr_min_stack ( void );
```

Поток может выяснить свой идентификатор через функцию *thr_self*:

```
thread_t    thr_self    ( void );
```

В приведенном ниже примере создается новый отсоединенный и связанный поток для выполнения функции *do_it*. Аргумент, передаваемый этой функции, представляет собой адрес переменной *pInt*. Идентификатор нового потока присваивается переменной *tid*; создается стек стандартного размера:

```
extern void* do_it(void* ptr);
int         *pInt;
thread_t    tid;
if (thr_create(0, 0, do_it, (void*)&pInt,
                        THR_DETACHEDITHR_BOUND, &tid) < 0)
    perror("thr_create");
```

13.3.2. Функции thr_suspend, thr_continue

Прототипы функций *thr_suspend* и *thr_continue* выглядят следующим образом:

```
#include <thread.h>

int    thr_suspend ( thread_t tid );
int    thr_continue ( thread_t tid );
```

Функция *thr_suspend* приостанавливает выполнение потока, идентификатор которого обозначен аргументом *tid*.

Функция *thr_continue* возобновляет выполнение потока, идентификатор которого обозначен аргументом *tid*.

Если аргумент *tid* имеет недопустимое значение, эти функции могут завершиться неудачно.

13.3.3. Функции thr_exit, thr_join

Прототипы функций *thr_exit* и *thr_join* выглядят следующим образом:

```
#include <thread.h>

int  thr_exit ( void* statusp );
int  thr_join ( thread_t tid, thread_t* dead_tidp, void** statusp );
```

Функция *thr_exit* завершает поток. Фактическим значением аргумента *statusp* является адрес статической переменной, которая содержит код возврата завершающегося потока. Если ни один из других потоков не будет использовать код возврата завершающегося потока (например, поток отсоединяется), значение этого аргумента можно указать как NULL.

Функция *thr_join* вызывается для ожидания завершения неотсоединенного потока. Если аргумент *tid* равен нулю, функция ожидает завершения любого потока. Значения аргументов *dead_tidp* и *status* — адреса переменных, которые содержат идентификатор и код возврата завершенного потока. Если эти данные не нужны, значения аргументов могут быть установлены равными NULL.

В приведенном ниже примере программа ждет завершения всех неотсоединенных потоков процесса, а затем завершает текущий поток:

```
status int *rc, rval = 0;
thread_t tid;
while (!thr_join(0, &tid, &rc))
    cout << "thread:" << (int)tid << ", exits, rc="<< (rc*) << endl;
thr_exit( (void*)&rval );
```

13.3.4. Функции thr_sigsetmask и thr_kill

У каждого потока есть сигнальная маска, которую он наследует от создавшего его процесса. Поток может модифицировать свою сигнальную маску, воспользовавшись функцией *thr_sigsetmask*. Когда в многопотоковый процесс поступает какой-то сигнал, его получает поток, в котором этот сигнал не заблокирован. Если сигнал не заблокирован в нескольких потоках, система выбирает в качестве пункта назначения сигнала один из этих потоков. Поэтому для упрощения реализации многопотоковых программ рекомендуется, чтобы процесс выбирал определенный поток для обработки одного и более сигналов, а в остальных потоках этого процесса данные сигналы блокировались.

Кроме того, поток может посылать сигнал в другой поток в этом же процессе, пользуясь функцией *thr_kill*. Прототипы функций *thr_sigsetmask* и *thr_kill* выглядят следующим образом:

```
#include <thread.h>
#include <signal.h>

int thr_sigsetmask ( int mode, sigset_t* sigsetp, sigset_t* oldsetp );
int thr_kill ( thread_t tid, int signum );
```

Функция *thr_sigsetmask* устанавливает сигнальную маску вызывающего потока. Аргумент *sigsetp* содержит один или несколько номеров сигналов, применяемых к вызывающему потоку. Аргумент *mode* показывает, как следует использовать сигнал (сигналы), заданный в аргументе *sigsetp*. Возможные значения аргумента *mode* объявляются в заголовке <signal.h>.

Значение аргумента *made*	Смысл
SIG_BLOCK	Добавляет сигналы, указанные в аргументе *sigsetp*, в сигнальную маску потока
SIG_UNBLOCK	Удаляет сигналы, указанные в аргументе *sigsetp*, из сигнальной маски потока
SIG_SETMASK	Заменяет сигнальную маску потока сигналом (или сигналами), указанным в аргументе *sigsetp*

Если аргумент *sigsetp* имеет значение NULL, значение аргумента *mode* игнорируется.

Значением аргумента *oldsetp* должен быть адрес переменной типа *sigset_t**, с помощью которой возвращается старая сигнальная маска. Если значение аргумента *oldsetp* — NULL, то старая сигнальная маска не возвращается.

Функция *thr_kill* передает сигнал, заданный аргументом *signum*, в поток, идентификатор которого задан аргументом *tid*. Передающий и принимающий потоки должны находиться в одном процессе.

В приведенном ниже примере программа добавляет сигнал SIGINT в сигнальную маску потока, а затем посылает сигнал SIGTERM в поток с идентификатором 15:

```
sigset set, oldset;
sigemptyset(&set);
sigaddset(&set, SIGINT);
if (thr_setsigmask(SIG_BLOCK, &set, &oldset))
perror("thr_sigsetmask");
if (thr_kill((thread_t)15, SIGTERM) perror("thr_kill");
```


13.3.5. Функции thr_setprio, thr_getprio и thr_yield

Прототипы функций *thr_setprio*, *thr_getprio* и *thr_yield* выглядят следующим образом:

```
#include <thread.h>

int   thr_setpria (thread_t tid, int prio);
int   thr_getpria (thread_t tid, int* priop);
void  thr_yield (void);
```

Функция *thr_setprio* устанавливает приоритет потока, обозначенного аргументом *tid*, в значение *prio*. Потоки с более высокими приоритетами назначаются к выполнению чаще, чем потоки с низкими приоритетами.

Функция *thr_getprio* возвращает текущее значение приоритета потока через аргумент *priop*. Этот поток обозначается аргументом *tid*.

Функция *thr_yield* вызывается потоком для передачи выполнения другим потокам с таким же приоритетом. Эта функция всегда выполняется успешно и никакого значения не возвращает.

Планирование потоков выполняется многопотоковыми библиотечными функциями, а не ядром. Потоки планируются для связывания с облегченными процессами, которые, в свою очередь, планируются ядром для выполнения на аппаратном процессоре.

13.3.6. Функции thr_setconcurrency и thr_getconcurrency

Прототипы функций *thr_setconcurrency* и *thr_getconcurrency* выглядят следующим образом:

```
#include <thread.h>

int   thr_setcancurrency (int amount);
int   thr_getcancurrency (void);
```

Функция *thr_setconcurrency* задает минимальное количество облегченных процессов, которые должны существовать в процессе. Таким образом обеспечивается возможность параллельного выполнения минимального числа потоков в любой момент времени. Заметим, что система принимает значение аргумента *amount* как рекомендацию, придерживаясь этого значения настолько, насколько позволяют наличные ресурсы.

Функция *thr_getconcurrency* возвращает значение, равное текущему количеству облегченных процессов для процесса.

13.3.7. Пример многопотоковой программы

Приведенная ниже программа — это измененная версия клиентской и серверной RPC-программ, рассмотренных в разделе 12.5. Программа-клиент получает от пользователя строку сообщения и выполняет RPC-вызов серверной функции *printmsg*, которая выводит сообщение на системную консоль сервера.

Изменения касаются только программы-клиента (*msg_cls.C*). Клиент многократно запрашивает у пользователя имя хост-сервера и сообщение. Клиентский процесс вызывает функцию, которая для каждого имени хост-машины и данных сообщения, принятого от пользователя, динамически выделяет область памяти. Затем процесс создает поток для передачи RPC-вызова серверу, работающему на указанной хост-машине, и требует, чтобы пользовательское сообщение было выведено на серверную системную консоль.

Ниже приведена новая программа-клиент *msg_cls2.C*:

```
/* client program: low-level RPC APIs */
#include <thread.h>
#include <signal.h>
#include "msg2.h"
#include "RPC.h"
extern "C" int   thr_sigsetmask(int, const sigset_t *, sigset_t *);

#define MAX_THREAD  200
/* запись для одной хост-машины и данные сообщения для одного потока */
typedef struct
{
    char   *host;
    char   *msg;
} MSGREC;

thread_t thread_list[MAX_THREAD];  // содержит идентификаторы всех
                                   // потоков

/* функция, с помощью которой поток передает сообщение */
void* send_msg( void* ptr )
{
    int  res;
    MSGREC *pRec = (MSGREC*)ptr;

    /* установить сигнальную маску потока в любое значение, кроме
       SIGHUP */
    sigset_t   setv;
    sigfillset(&setv);
    sigdelset(&setv, SIGHUP);
    if (thr_sigsetmask(SIG_SETMASK,&setv,0)) perror("thr_setsigmask");

    /* создать обработчик клиента для связи с хост-машиной */
    RPC_cls cl( pRec->host, MSGPROG, MSGVER, "netpath");
    if (!cl.good()) thr_exit( &res );
```



```
    /* вызвать удаленную хост-машину для вывода сообщения на ее
       системную консоль */
    (void)cl.call( PRINTMSG, (xdrproc_t)xdr_string, (caddr_t)&(pRec->msg),
            (xdrproc_t)xdr_int, (caddr_t)&res);

    /* освободить динамически выделяемую память */
    delete pRec->msg;
    delete pRec->host;
    delete pRec;

    /* проверить статус выполнения RPC-функции */
    if (res!=0) cerr << "clnt: call printmsg falls\n";
    int *rcp = new int(res);
    thr_exit( rcp );
    return 0;
}

/* функция, которая создает поток для пользовательского сообщения */
int add_thread( int& num_thread )
{
    char host[60], msg(256];
    thread_t tid;
    int res;

    /* получить от пользователя имя удаленной хост-машины и сообщение */
    cin >> host >> msg;
    if (cin.eof()) return RPC_FAILED;   /* нормальное завершение */
    if (!cin.good()) {          /* обнаружена ошибка ввода-вывода */
        perror("cin");
        return RPC_FAILED;
    }

    /* выделить память для текста сообщения и имени хост-машины*/
    MSGREC *pRec = new MSGREC;
    pRec->host = new char[strlen(host)+1];
    pRec->msg = new char[strlen(msg)+1];
    strcpy(pRec->host,host);
    strcpy(pRec->msg,msg);

    /* для обработки сообщения создать ожидающий выполнения поток */
    if (thr_create( 0, 0, send_msg, pRec, THR_SUSPENDED, &tid ))
    {
        perror("thr_create");
        return RPC_FAILED;
    }
    else if (num_thread>= MAX_THREAD)
    {
        cerr << "Too many threads created!\n";
        return RPC_FAILED;
    }
    else {/* создать поток */
```

```
        thread_list[num_thread++] = tid;
        cout << "Thread: " << (int)tid << " created for msg: '"
             << msg << " [" << host << "]\n";
    }
    return RPC_SUCCESS;
}

/* Главная функция клиента */
int main(int argc, char* argv[])
{
    int     num_thread=0;
    thread_t  tid;
    int     *res;

    /* задать количество параллельных потоков */
    if (thr_setconcurrency(5)) perror("thr_setoncurrency");
    cout << "No. LPWs in process: " << getpid() << " is: "
         << thr_getconcurrency() << endl;

    /* создать поток для передачи каждого сообщения */
    while (add_thread(num_thread)==RPC_SUCCESS) ;

    /* установить приоритет для каждого потока и запустить его */
    for (int i=0; i < num_thread; i++)
    {
        thr_setprio(thread_list[i],i);
        thr_continue(thread_list[i]);
    }

    /* ждать завершения каждого потока */
    while (!thr_join(0,&tid,(void**)&res))
    {
       cerr << "thread: " << (int)tid
            << ", exited. rc=" << (*res) << endl;
       delete res;
    }
    /* завершить главный поток */
    thr_exit(0);
    return 0;
}
```

В этой программе клиентский процесс начинается с вызова функции *thr_concurrency*, которая устанавливает число облегченных процессов равным пяти. Это значит, что в любой момент времени параллельно могут выполняться как минимум пять потоков. Процесс выясняет количество фактически созданных LWP с помощью функции *thr_getconcurrency* и отображает эту информацию на стандартном устройстве вывода.

Затем процесс вызывает функцию *add_thread* до тех пор, пока она не возвратит нуль. Каждый раз при вызове эта функция получает от пользователя через стандартный ввод имя хост-машины и сообщение. Если встречается признак конца файла или ошибка ввода, функция возвращает в *main*

нулевое значение, означающее окончание пользовательского ввода. Если входные данные выбраны успешно, функция *add_thread* выделяет память под хранение указанного пользователем имени хост-машины и сообщения в записи типа MSGREC. Затем она вызывает функцию *thr_create* для создания потока и выполнения функции *send_msg*. Входным аргументом функции *send_msg* является адрес только что определенной переменной типа MSGREC. Этот поток создается со стеком, выделенным системой, и приостанавливается сразу же после создания. Идентификатор вновь созданного потока сохраняется в глобальном массиве *thread_list*, и функция *add_thread* возвращает в *main* единицу, указывая таким образом на успешное выполнение. Если же поток не создался или если уже создано MAX_THREAD потоков, функция возвращает в *main* нулевое значение, сигнализирующее об ошибке.

После того как функция *add_thread* создаст все потоки, необходимые для обработки всех пользовательских входных данных, функция *main* (главный поток) просматривает массив *thread_list* и устанавливает приоритет каждого потока. Идентификатор потока сохраняется в массиве со значением, связанным с позицией (индексом) потока в массиве. Так, первый поток в массиве имеет самый низкий приоритет, следующий поток имеет второй приоритет снизу и так далее. Функция *main* запускает каждый приостановленный поток с помощью функции *thr_continue*. После этого функция *main* ждет завершения каждого потока и отображает на стандартном устройстве вывода идентификатор потока и код возврата.

Каждый запущенный поток выполняет функцию *send_msg*. Сначала поток устанавливает свою сигнальную маску так, чтобы она включала все, кроме сигнала SIGHUP. Затем поток создает обработчик RPC-клиента, который будет соединяться с хост-сервером, чье имя задано во входном аргументе функции *send_msg*. Если обработчик RPC-клиента создан успешно, то поток вызывает функцию *RPC_cls::call*, которая передает пользовательское сообщение на сервер. Сообщение выводится на системную консоль сервера. Наконец, поток освобождает память, в которой хранится имя хост-машины и сообщение, выделяет память для хранения возвращенного значения, после чего с помощью вызова *thr_exit* возвращает эти динамические данные в функцию *main*. Код возврата хранится в динамически выделяемой памяти по той причине, что функция *send_msg* выполняется множеством потоков и каждый из них должен возвращать собственный код завершения. Поэтому возвращенное значение не может храниться в статической переменной, а должно помещаться в динамическую переменную, уникальную для каждого потока.

Программа-сервер *printmsg* (*msg_svc2.C*) и определение класса *RPC_cls*, указанного в заголовке *RPC.h*, приведены в разделе 12.5. Новая программа-клиент *msg_cls2.C* компилируется следующим образом:

```
% CC -DSYSV4 msg_cls2.C -o msg_cls2 -lthread -lnsl
```

Опции *-lthread* и *-lnsl* дают редактору связей указание скомпоновать объектный файл *msg_cls2.o* с библиотекой потоков выполнения (*libthread.so*) и сетевой библиотекой (*libnsl.co*). Первая библиотека содержит объектные коды всех API потоков выполнения, а вторая — коды, связанные с RPC.

Ниже показаны пример запуска клиентской программы и полученные результаты. Слова, выделенные курсивом,— это входные данные, вводимые пользователем со стандартного устройства ввода.

```
% msg_cls2
No. LWPs: 5
fruit   happy_day
Thread: 4 created for msg: 'happy_day [fruit]'
fruit   easter_sunday
Thread: 5 created for msg: 'easter_sunday [fruit]'
fruit   Good_bye
Thread: 6, created for msg: 'Good_bye [fruit]'
thread: 5, exited.rc=139424
thread: 6, exited.rc=139424
thread: 4, exited.rc=139424
%
```

На системной консоли машины *fruit* при этом появляются следующие сообщения:

```
server:'easter_sunday'
server:'Good_bye'
server:'happy_day'
```

13.4. API потоков выполнения, определенный в стандарте POSIX.1c

В этом разделе речь пойдет об API, который предусмотрен в стандарте POSIX.1c для основных операций управления потоками. Ниже приведена таблица соответствия API потоков выполнения стандарта POSIX.1c и фирмы Sun.

API потоков выполнения Sun	API потоков выполнения POSIX.1c
thr_create	pthread_create
thr_self	pthread_self
thr_exit	pthread_exit
thr_kill	pthread_kill
thr_join	pthread_join

Чтобы программа могла использовать эти API, в нее должен быть включен заголовок <pthread.h>, в котором объявляются все прототипы многопотоковых функций стандарта POSIX.1c. Кроме того, если программа управляет приоритетами планирования потоков, необходимо включить и заголовок <sched.h>.

13.4.1. Функция pthread_create

Прототип функции *pthread_create* выглядит следующим образом:

```
#include <pthread.h>

int pthread_create (pthread_t* tid_p, const pthread_attr_t* attr,
                    void* (*funcp)(void*), void* argp);
```

Эта функция создает новый поток для выполнения функции, адрес которой задан аргументом *funcp*. Функция, указанная в этом аргументе, должна принимать один входной аргумент типа *void** и возвращать данные такого же типа. Фактический аргумент, который передается в функцию *funcp* в начале выполнения нового потока, указывается в аргументе *argp*.

Идентификатор нового потока возвращается через аргумент *tid_p*. Если этому аргументу присвоено значение NULL, идентификатор не возвращается. Тип данных идентификатора потока — *pthread_t*.

Аргумент *attr* содержит атрибуты, присваиваемые вновь создаваемому потоку. Значение этого аргумента может быть равно NULL, если новый поток должен использовать атрибуты, принятые в системе по умолчанию, или адресу объекта, содержащего атрибуты. В POSIX.1c определяется набор API для создания, удаления, запроса и установки атрибутов. Объект, содержащий атрибуты, может быть связан с несколькими потоками, чтобы при каждом обновлении атрибутов сделанные изменения распространялись на все потоки, связанные с этим объектом. Напомним, что для каждого Sun-потока атрибуты задаются индивидуально.

Функция *pthread_attr_init* создает объект, содержащий атрибуты, а функция *pthread_attr_destroy* удаляет такой объект:

```
#include <pthread.h>

int pthread_attr_init( pthread_attr_t* attr_p );
int pthread_attr_destroy( pthread_attr_t* attr+p );
```

Атрибуты объекта, созданного функцией *pthread_attr_init*, можно проверить функцией *pthread_attr_get* или установить функцией *pthread_attr_set*. Ниже перечислены атрибуты, которые могут содержаться в объекте, и соответствующие API для их проверки и установки.

Атрибут	API для проверки	API для установки
Область действия конкуренции	pthread_attr_getscope	pthread_attr_setscope
Размер стека	pthread_attr_getstacksize	pthread_attr_setstacksize

Атрибут	API для проверки	API для установки
Адрес стека	pthread_attr_getstackaddr	pthread_attr_setstackaddr
Состояние отсоединения	pthread_attr_getdetachstate	pthread_attr_setdetachstate
Правила планирования	pthread_attr_getschedpolicy	pthread_attr_setschedpolicy
Параметры планирования	pthread_attr_getschedparam	pthread_attr_setschedparam

Все эти API *pthread_attr_get* принимают два аргумента: указатель на объект, содержащий атрибуты, и адрес переменной, в которой должно храниться затребованное значение атрибута. Все API *pthread_attr_set* также принимают два аргумента: указатель на объект и либо новое значение атрибута, либо указатель на переменную, где хранится это новое значение.

Область действия конкуренции при планировании рассматривалась в разделе 13.2. Возможные значения этого атрибута — PTHREAD_SCOPE _PROCESS и PTHREAD_SCOPE_SYSTEM.

Состояние отсоединения потока показывает, каким создается поток: отсоединенным или присоединяемым. Возможные значения — PTHREAD_ CREATE_DETACHED или PTHREAD_CREATE_JOINABLE.

Правила планирования потока задают, среди прочего, приоритет потока. Второй аргумент в API *pthread_attr_getschedparam* и *pthread_attr_setsched-param* — это адрес переменной типа *struct sched_param*. В этой переменной есть целочисленное поле *shced_priority*, в котором задается приоритет любого потока, обладающего этим свойством.

Наконец, размер и адрес динамического стека вновь создаваемого потока можно устанавливать почти так же, как в аргументах *stack_size* и *stackp* при вызове функции *thr_create* (см. раздел 13.3.1). Однако для установки этих свойств для одного или нескольких потоков используются API *pthread _attr_setstacksize* и *pthread_attr_setstackaddr*.

В приведенном ниже примере создается новый отсоединенный и связанный поток с приоритетом 5. Этот поток выполняет функцию *do_it* с аргументом, заданным переменной *pInt*. Идентификатор нового потока присваивается переменной *tid*, а выделяемый стек имеет стандартный размер:

```
extern void* do_it(void* ptr);
int                 *pInt;
pthread_t           tid;
pthread_attr_t      attr, *attrPtr = &attr;
struct shed_param   sched;

if (pthread_attr_init(attrPtr) == -1) {
    perror("pthread_attr_init");
    attrPtr = 0;
}
else {
    pthread_attr_setdetachstate( attrPtr, PTHREAD_CREATE_DETACH );
    pthread_attr_setscope( attrPtr, PTHREAD_SCOPE_SYSTEM );
    if (pthread_attr_getschedparam(attrPtr, &sched)==0) {
```



```
        param.sched_priority = 5;
        pthread_attr_setschedparam(attrPtr, &sched);
    }
}
if (pthread_create(&tid, &attr, do_it, (void*)&pInt)== -1)
    perror("pthread_create");
```

13.4.2. Функции pthread_exit, pthread_detach и pthread_join

Прототипы функций *pthread_exit*, *pthread_detach* и *pthread_join* приведены ниже:

```
#include <pthread.h>

int   pthread_exit ( void* status );
int   pthread_detach (void* status );
int   pthread_join ( pthread_t tid, void** statusp );
```

И функция *pthread_exit*, и функция *pthread_detach* завершают поток. Функция *pthread_exist* может использоваться неотсоединенным потоком, а функция *pthread_detach* применяется отсоединенным потоком. Значение аргумента *statusp* — адрес статической переменной, которая содержит код возврата завершающегося потока. Если никакой другой поток не будет использовать код возврата завершающегося потока, значение этого аргумента можно указать как NULL.

Функция *pthread_join* вызывается для ожидания завершения неотсоединенного потока. Она возвращает код возврата потока, переданный через вызов функции *pthread_exit*.

В приведенном ниже примере программа ожидает завершения всех неотсоединенных потоков процесса, а затем завершает текущий поток:

```
status int        *rc, rval = 0;
thread_t          tid;
if (!pthread_join(tid, &rc)
      cout << "thread:"<< (int)tid <<",exits, rc=" << (*rc) <<endl;
pthread_exit((void*)&rval);
```

13.4.3. Функции pthread_sigmask и pthread_kill

Прототипы функций *pthread_sigmask* и *pthread_kill* имеют вид:

```
#include <pthread.h>
#include <signal.h>

int   pthread_sigmask ( int mode, sigset_t* sigsetp, sigset_t* oldsetp );
int   pthread_kill ( pthread_t tid, int signum );
```

Функция *pthread_sigmask* устанавливает сигнальную маску вызывающего потока. Аргумент *sigsetp* содержит один или несколько номеров сигналов, применяемых к вызывающему потоку. Аргумент *mode* показывает, как следует использовать сигнал (сигналы), заданный в аргументе *sigsetp*. Возможные значения аргумента *mode* объявляются в заголовке <signal.h>:

Значение аргумента *mode*	Смысл
SIG_BLOCK	Добавляет сигналы, указанные в аргументе *sigsetp*, в сигнальную маску потока
SIG_UNBLOCK	Удаляет сигналы, указанные в аргументе *sigsetp*, из сигнальной маски потока
SIG_SETMASK	Заменяет сигнальную маску потока сигналом (или сигналами), указанным в аргументе *sigsetp*

Если аргумент *sigsetp* имеет значение NULL, значение аргумента *mode* игнорируется.

Значением аргумента *oldsetp* должен быть адрес переменной типа *sigset_t**, в которой возвращается старая сигнальная маска. Если значение аргумента *oldsetp* — NULL, то старая сигнальная маска не возвращается.

Функция *pthread_kill* посылает сигнал, заданный аргументом *signum*, в поток, идентификатор которого задан аргументом *tid*. Передающий и принимающий потоки должны находиться в одном процессе.

Ниже приведен пример программы, которая добавляет сигнал SIGINT в сигнальную маску потока, а затем посылает сигнал SIGTERM в поток с идентификатором 15:

```
sigset_t set, oldset;
sigemptyset(&set);
sigaddset(&set, SIGINIT);
if (pthread_setmask(SIG_BLOCK, &set, &oldset))
    perror("pthread_sigsetmask");
if (pthread_kill(thread_t)15, SIGTERM)) perror("pthread_kill");
```

13.4.4. Функция sched_yield

Ниже приведен прототип функции *sched_yield*.

```
#include <pthread.h>

int  sched_yield (void);
```

Поток вызывает функцию *sched_yield*, чтобы передать выполнение другим потокам с таким же приоритетом. При успешном выполнении данная функция возвращает 0, а в случае неудачи — -1. Этот API стандарта POSIX.1c эквивалентен API *thr_yield* фирмы Sun.

13.5. Объекты синхронизации потоков выполнения

Потоки выполнения в процессе совместно используют то же адресное пространство, что и процесс. Это значит, что все глобальные и статические переменные процесса доступны для всех его потоков. Чтобы обеспечить безошибочное манипулирование этими переменными, потоки должны как-то синхронизировать свою работу. В частности, ни один поток не должен обращаться к переменной, значение которой в текущий момент изменяется другим потоком, и ни один поток не может изменять значение переменной в то время, когда его читают другие потоки.

Синхронизация нужна и тогда, когда два и более потока выполняют операции с общим потоком ввода-вывода. Например, если два потока выполнения одновременно записывают информацию в поток вывода, то невозможно предсказать, какие именно данные попадут в этот поток. Аналогичная проблема возникает при одновременном чтении данных из потока ввода.

Чтобы разрешить проблемы синхронизации, Sun и POSIX.1c предлагают для управления операциями, которые производят потоки выполнения над общими данными и потоками ввода-вывода в процессе, использовать следующие объекты:

- взаимоисключающие блокировки;
- условные переменные;
- семафоры.

Самые примитивные и самые эффективные из этих объектов — взаимоисключающие блокировки. Они применяются для организации последовательного доступа к общим данным и выполнения сегментов кода.

Второй по эффективности объект — условные переменные. Как правило, они используются с взаимоисключающими блокировками для управления асинхронным доступом к общим данным.

Семафоры более сложны, нежели взаимоисключающие блокировки и условные переменные. Они используются почти так же, как семафоры UNIX System V и POSIX.1b.

Помимо этих средств, Sun предлагает еще один инструмент синхронизации потоков выполнения — блокировки чтения-записи. Такие блокировки обеспечивают множественный доступ для чтения и однократный доступ для записи к любым общим данным. Перечисленные выше объекты такой способностью не обладают. Блокировки чтения-записи используются прежде всего для защиты данных, которые часто читаются многими потоками, но редко изменяются.

Процессы, которые применяют какие-либо объекты синхронизации, должны определить для них область памяти в своем виртуальном адресном пространстве. Если эти объекты определяются в общих областях памяти, которые доступны для многих процессов, их можно использовать для синхронизации потоков выполнения в этих процессах.

Рассмотрим объекты синхронизации подробнее.

13.5.1. Взаимоисключающие блокировки

Взаимоисключающие блокировки позволяют организовать выполнение потоков так, что когда несколько потоков пытаются установить такую блокировку, успеха достигает только один из них, который и продолжает выполняться. Остальные потоки блокируются до тех пор, пока эта блокировка не будет снята потоком-владельцем. При снятии взаимоисключающей блокировки невозможно предсказать, какой ожидающий поток будет следующим владельцем блокировки.

В приведенной ниже таблице представлены API взаимоисключающих блокировок, разработанные фирмой Sun, а также API взаимоисключающих блокировок стандарта POSIX.1c.

API Sun	API POSIX.1c
mutex_init	pthread_mutex_init
mutex_destroy	pthread_mutex_destroy
mutex_lock	pthread_mutex_lock
mutex_trylock	pthread_mutex_trylock
mutex_unlock	pthread_mutex_unlock

Эти API более подробно рассматриваются ниже.

13.5.1.1. Взаимоисключающие блокировки Sun

Фирма Sun использует для операций с взаимоисключающими блокировками следующие многопотоковые библиотечные функции:

Функция	Назначение
mutex_init	Инициализирует взаимоисключающую блокировку
mutex_lock	Устанавливает взаимоисключающую блокировку
mutex_unlock	Снимает взаимоисключающую блокировку
mutex_trylock	Как *mutex_lock*, только неблокирующая
mutex_destroy	Удаляет взаимоисключающую блокировку

Эти функции имеют следующие прототипы:

```
#include <thread.h>

int   mutex_init (mutex_t* mutxp, int type, void* argp);
int   mutex_lock (mutex_t* mutxp);
int   mutex_trylock (mutex_t* mutxp);
int   mutex_unlock (mutex_t* mutxp);
int   mutex_destroy (mutex_t* mutxp);
```


Значение аргумента *mutxp* — адрес переменной типа *mutex_t*. Эта переменная определяется вызывающим потоком и устанавливается как ссылка на взаимоисключающую блокировку с помощью функции *mutex_init*. Аргумент *type* функции *mutex_init* показывает, доступна ли данная блокировка для потоков в других процессах. Ниже приведены его возможные значения:

Значение аргумента *type*	Смысл
USYNC_PROCESS	Взаимоисключающая блокировка может использоваться потоками в других процессах
USYNC_THREAD	Взаимоисключающая блокировка может использоваться потоками только в вызывающем процессе

Аргумент *argp* функции *mutex_init* в настоящее время не задействуется, поэтому его значение должно быть равно 0.

В процессе взаимоисключающая блокировка инициализируется только один раз и удаляется функцией *mutex_destroy*.

Взаимоисключающая блокировка устанавливается потоком с помощью функции *mutex_lock*, а снимается функцией *mutex_unlock*. Функция *mutex_lock* блокирует вызывающий поток, если взаимоисключающая блокировка уже установлена другим потоком. Когда эта взаимоисключающая блокировка снимается, поток разблокируется и после этого может стать владельцем новой блокировки.

Функция *mutex_trylock* отличается от функции *mutex_lock* тем, что, если затребованной блокировкой уже владеет другой поток, функция возвращает в вызывающий поток код ошибки, но не блокирует его.

13.5.1.2. Взаимоисключающие блокировки POSIX.1c

API для операций со взаимоисключающими блокировками, предусмотренные в стандарте POSIX.1c, похожи на соответствующие API Sun:

Функция	Назначение
pthread_mutex_init	Инициализирует взаимоисключающую блокировку
pthread_mutex_lock	Устанавливает взаимоисключающую блокировку
pthread_mutex_unlock	Снимает взаимоисключающую блокировку
pthread_mutex_trylock	Как *mutex_lock*, только неблокирующая
pthread_mutex_destroy	Удаляет взаимоисключающую блокировку

Эти функции имеют следующие прототипы:

```
#include <pthread.h>

int   pthread_mutex_init (pthread_mutex_t* mutxp, pthread_mutexattr_t* attrp);
int   pthread_mutex_lock (pthread_mutex_t* mutxp);
int   pthread_mutex_trylock (pthread_mutex_t* mutxp);
int   pthread_mutex_unlock (pthread_mutex_t* mutxp);
int   pthread_mutex_destroy (pthread_mutex_t* mutxp);
```

Значение аргумента *mutxp* — адрес переменной типа *pthread_mutex_t*. Эта переменная определяется вызывающим потоком и устанавливается как ссылка на взаимоисключающую блокировку с помощью функции *pthread_mutex_init*. Аргумент *attrp* — это указатель на объект, содержащий атрибуты для новой взаимоисключающей блокировки.

В качестве альтернативы вызову *pthread_mutex_init* может использоваться статический инициализатор PTHREAD_MUTEX_INITIALIZER. Таким образом, следующий код

```
pthread_mutex_t        lockx;
(void)pthread_mutex_init(&lockx, 0);
```

идентичен оператору

```
pthread_mutex_t        lockx = PTHREAD_MUTEX_INITIALIZER;
```

Синтаксис вызова и назначение функций *pthread_mutex_lock*, *pthread_mutex_trylock*, *pthread_mutex_unlock*, *pthread_mutex_destroy* такие же, как и у API Sun *mutex_lock*, *mutex_trylock*, *mutex_unlock* и *mutex_destroy*.

13.5.1.3. Примеры взаимоисключающих блокировок

Показанная ниже функция *printmsg* может вызываться любым потоком для отображения сообщений на стандартном устройстве вывода. Она гарантирует вывод сообщений потоков по одному. Функция *main* — это пример приложения, тестирующего функцию *printsmg*:

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

static mutex_t lockx;        // определить переменную для блокировки
void* printmsg (void* msg )
{
    /* установить блокировку */
    if (mutex_lock(&lockx))
        perror("mutex_lock");
    else {
        /* вывести сообщение */
        cout << (char*)msg << endl << flush;

        /* снять блокировку */
        if (mutex_unlock(&lockx)) perror("mutex_unlock");
    }
    return 0;
}

int main(int argc, char* argv[]);
{
    /* инициализировать взаимоисключающую блокировку */
    if (mutex_init(&lockx, USYNC_THREAD,NULL))
        perror("mutex_lock");
    /* создать потоки, которые вызывают printmsg */
```



```
    while (--argc > 0)
        if (thr_create(0,0,printmsg,argv[argc],0, 0))
            perror("thr_create");

    /* ждать завершения всех потоков */
    while (!thr_join(0,0,0)) ;

    /* удалить взаимоисключающую блокировку */
    if (mutex_destroy(&lockx)) perror("mutex_destroy");

    return 0;
}
```

Функция *main* в этом примере инициализирует взаимоисключающую блокировку *lockx*, которой будут пользоваться все потоки в этом процессе. Затем функция создает несколько потоков — по одному для каждого аргумента командной строки,— которые будут вызывать функцию *printmsg* и отображать на стандартном устройстве вывода строки аргументов команд. Количество потоков, одновременно вызывающих функцию *printmsg*, не имеет значения, потому что эта функция будет обслуживать потоки по одному.

Ниже приведены пример запуска и результаты работы программы *printmsg.C*:

```
% CC printmsg.C -lthread -o printmsg
% printmsg "1" "2" "3"
3
2
1
```

А теперь рассмотрим еще один пример — класс *glob_dat*, определяющий тип данных для переменных, к которым одновременно могут обращаться несколько потоков. Этот класс определяется в заголовке *glob_dat.h*:

```
#ifndef GLOB_DAT_H
#define GLOB_DAT_H

#include <thread.h>
#include <iostream.h>
#include <stdio.h>

class glob_dat
{
    private:
        int         val;
        mutex_t     lockx;
    public:
        /* функция-конструктор */
        glob_dat( int a )
        {
            val = a;
            if (mutex_init(&lockx, USYNC_THREAD,0))
                perror("mutex_init");
        };
        /* функция-деструктор */
```

```
    ~glob_dat() { if (mutex_destroy(&lockx))
                  perror("mutex_destroy"); );

    /* установить новое значение */
    gl ob_dat& operator=( int new_val )
    {
        if (!mutex_lock(&lockx)) {
            val = new_val;
            if (mutex_unlock(&lockx)) perror("mutex_unlock");
        )else perror("mutex_lock");

        return *this;

    };

    /* выбрать значение */
    int getval( int* valp )
    {
        if (!mutex_lock(&lockx)) {
            *valp= val;
            if (!mutex_unlock(&lockx)) return 0;
            //perror("mutex_unlock");
        } else perror("mutex_lock");

        return -1;
    };

    /* направить значение в поток вывода */
    friend ostream& operator<<( ostream& os, glob_dat& gx)
    {
        lf (!mutex_lock(&gx.lockx)) {
            os << gx.val;
            if (mutex_unlock(&gx.lockx)) perror("mutex_lock");

            } else     perror("mutex_lock");

        return os;

    };

};          /* glob_dat */
#endif
```

Значение переменной типа *glob_dat* сохраняется в поле *glob_dat::val*. Взаимоисключающая блокировка *glob_dat::lockx* используется для организации доступа к *glob_dat::val* для нескольких процессов. Члены *glob_dat::lockx* и *glob_dat::val* инициализируются при определении переменной этого типа. Блокировка *glob_dat::lockx* удаляется при вызове функции-деструктора *glob_dat::~glob_dat*.

Функции-члены *glob_dat::getval*, *glob_dat::show* и *glob_dat::operator=* позволяют потокам выбирать, показывать и устанавливать любое значение переменной типа *glob_dat*. Благодаря наличию взаимоисключающей блокировки

эти функции могут одновременно вызываться несколькими потоками для работы с одной и той же переменной.

Приведенный ниже пример (программа *glob_dat.C*) показывает, как несколько потоков обращаются к переменной типа *glob_dat* и *globx*:

```
#include <strstream.h>
#include <thread.h>
#include "glob_dat.h"

glob_dat globx (0);                         // определить globx

void* mod_val (void* np )
{
    int old_val=0, new_val;
    istrstream((char*)np) >> new_val;

    if (!globx.getval(&old_val)) {          // получить текущее значение
        globx = old_val + new_val;          // добавить новое значение
        cout << (int)thr_self() << ": arg='" << (char*)np
             << "'-> " << globx << endl; // показать результат
    }
    return 0;
}

int main( int argc, char*argv[])
{
    thread_t tid;
    while (--argc > 0)     // для каждого аргумента командной строки
        if (thr_create(0,0,mod_val,argv[argc],0,&tid))
        perror("thr_create");
    while (thr_join(0,0,0)) ;     // ожидать завершения потоков
    return 0;
}
```

Функция *main* создает переменную *globx* с нулевым начальным значением. Затем для каждого аргумента командной строки она создает поток выполнения функции *mod_val*. С каждым вызовом *mod_val* в качестве аргумента передается аргумент командной строки. Он состоит из текстовой строки целочисленного значения (например, "51"). Функция *mod_val* преобразует свой аргумент в целочисленное значение, получает текущее значение *globx* и добавляет новое целочисленное значение к текущему. Результат присваивается переменной *globx*, значение которой затем отображается на стандартном устройстве вывода.

Ниже приведен пример запуска этой программы и полученные результаты:

```
% CC glob_dat.C -lthread -o glob_dat
% glob_dat "1" "2" "3"
4: arg='3' -> 3
5: arg='2' -> 5
6: arg='1' -> 6
```

При использовании взаимоисключающих блокировок следует избегать создания тупиковых ситуаций. Такая ситуация может возникнуть, когда процесс использует несколько взаимоисключающих блокировок и два или более потоков в этом процессе пытаются стать владельцами этих блокировок в произвольной последовательности. Пусть, например, в процессе инициализированы две взаимоисключающие блокировки (A и B), которыми пользуются два потока (X и Y). Допустим, поток X установил блокировку A, а поток Y — блокировку B. Когда X пытается стать владельцем блокировки B, он блокируется, потому что эта блокировка установлена потоком Y. Если затем поток Y пытается стать владельцем блокировки A, он тоже блокируется. Таким образом создается тупиковая ситуация: блокируются оба потока, потому что каждый из них пытается установить блокировку, принадлежащую другому потоку. Ни один из этих потоков не будет выполняться, потому что владелец блокировки заблокирован и не может снять ее.

Во избежание таких ситуаций потоки, пользующиеся взаимоисключающими блокировками, должны всегда устанавливать их в одном и том же порядке и/или пользоваться для установки блокировок функцией *thr_trywait*, а не функцией *thr_wait*. В первом случае все потоки устанавливают блокировки в одном и том же порядке, поэтому поток не может попытаться завладеть блокировкой, принадлежащей заблокированному потоку. Если же поток устанавливает блокировку с помощью функции *thr_trywait*, то она сразу же возвращает ненулевое значение, показывающее, что блокировка принадлежит другому потоку. Поток, таким образом, не блокируется, может выполнять другие операции и повторить попытку установки блокировки позже.

13.5.2. Условные переменные

Условные переменные используются для того, чтобы заблокировать потоки до выполнения определенных условий. Условные переменные обычно применяются в сочетании со взаимоисключающими блокировками, чтобы несколько потоков могли ожидать момента выполнения одного условия. Это можно сделать следующим образом. Сначала поток устанавливает взаимоисключающую блокировку, но и сам блокируется до момента выполнения определенного условия. На то время, пока поток заблокирован, установленная им блокировка автоматически снимается. Когда другой поток выполняет поставленное условие, он дает условной переменной сигнал о разблокировании первого потока. После разблокирования потока взаимоисключающая блокировка автоматически вновь устанавливается, и первый поток повторно проверяет условие. Если оно не выполняется, поток опять блокируется условной переменной. Если же условие выполняется, поток снимает взаимоисключающую блокировку и выполняется дальше.

13.5.2.1. Условные переменные фирмы Sun

Ниже перечислены многопотоковые библиотечные функции фирмы Sun для управления условными переменными.

Функция	Назначение
cond_init	Инициализирует условную переменную
cond_wait	Блокирует по условной переменной
cond_timedwait	То же, что *cond_wait*, но указывается тайм-аут блокировки
cond_signal	Разблокирует поток, ожидающий условную переменную
cond_broadcast	Разблокирует все потоки, ожидающие условную переменную
cond_destroy	Удаляет условную переменную

Прототипы этих функций выглядят следующим образом:

```
#include <thread.h>

int cond_init (cond_t* condp, int type, int argp);
int cond_wait (cond_t* condp, mutex_t* mutxp,);
int cond_timedwait (cond_t* condp, mutex_t* mutxp, timestruct_t* timp);
int cond_signal (cond_t* condp);
int cond_broadcast (cond_t* condp);
int cond_destroy (cond_t* condp);
```

Значение аргумента *condp* — адрес переменной типа *cond_t*. Эта переменная устанавливается как ссылка на условную переменную посредством функции *cond_init*. Аргумент *type* функции *cond_init* показывает, доступна ли данная условная переменная для других процессов. Приведем его возможные значения.

Значение аргумента *type*	Смысл
USYNC_PROCESS	Условная переменная может использоваться потоками в других процессах
USYNC_THREAD	Условная переменная может использоваться потоками только в вызывающем процессе

Аргумент *arg* функции *cond_init* в настоящее время не задействуется. Его значение должно быть равно 0.

Функция *cond_wait* блокирует вызывающий поток, заставляя его ожидать изменения условной переменной в соответствии с аргументом *condp*. Она также снимает взаимоисключающую блокировку, заданную аргументом *mutxp*.

Когда другой поток вызывает функцию *cond_signal* по той же условной переменной, взаимоисключающая блокировка вновь устанавливается ожидающим потоком. Сам этот поток разблокируется и возобновляет выполнение

после возврата из функции *cond_wait*. Если эта условная переменная заблокировала несколько потоков, все они должны использовать одну и ту же взаимоисключающую блокировку. Когда условная переменная изменяет состояние, установить взаимоисключающую блокировку и продолжать выполнение может только один из заблокированных потоков. Остальные потоки остаются заблокированными этой же блокировкой.

Функция *cond_timewait* похожа на функцию *cond_wait*, только в ней есть третий аргумент, *timp*, который показывает, что вызывающий поток разблокируется по истечении времени, указанного в этом аргументе.

Функция *cond_signal* изменяет состояние условной переменной, заданной аргументом *condp*, и разблокирует ожидающий эту переменную поток. Функция *cond_broadcast* изменяет состояние условной переменной и разблокирует все ожидающие потоки. Если данную условную переменную не ожидает ни один поток, то вызовы *cond_signal* и *cond_broadcast* с этой переменной не влекут никаких последствий.

13.5.2.2. Пример программы с условными переменными

Приведенная ниже программа *pipe.C* иллюстрирует использование взаимоисключающей блокировки и условной переменной. Эта программа создает поток чтения и поток записи. Поток чтения читает данные, вводимые пользователем, и передает их в поток записи через глобальный массив *msgbuf*. Поток записи направляет сообщения, содержащиеся в *msgbuf*, на стандартное устройство вывода. Эти два потока завершаются, когда во входных данных встречается признак конца файла. Доступ потоков чтения и записи к массиву *msgbuf* синхронизируется с помощью взаимоисключающей блокировки и условной переменной.

Вот текст программы *pipe.C*:

```
#include <iostream.h>
#include <thread.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>

#define FINISH()    { cerr << (int)thr_self() << " exits\n"; \
                      mutex_unlock(&mutx); thr_exit(0); return 0; }

mutex_t mutx;
cond_t  condx;
int     msglen, done;
char    msgbuf[256];

/* направить сообщения, посылаемые из потока чтения, на стандартное
   устройство вывода */
void* writer (void* argp )
{
    do {
        mutex_lock(&mutx); // установить взаимоисключающую блокировку
        while (!msglen) (  // цикл, если сообщения нет
            cond_wait(&condx,&mutx); // ждать условную переменную
            if (done) FINISH(); // если выполнено, уничтожить поток
        )
        cout << "*> " << msgbuf << endl;  /* направить сообщение на
                                            стандартное устройство
                                            вывода */
        msglen = 0;                       // размер буфера сообщений
        mutex_unlock(&mutx); // снять взаимоисключающую блокировку
    ) while (1);

    FINISH();                             // завершение и выход
}

/* читать сообщения пользователя и посылать их в поток записи */
void* reader (void* argp )
(
    do (
        mutex_lock(&mutx); // установить взаимоисключающую блокировку
        if (!msglen) (     // проверить, пуст ли буфер
            if (!cin.getline(msgbuf,256)) break; /* получить входные
                                                    данные от
                                                    пользователя */
            //if (cin.eof()) break;
            msglen = strlen(msgbuf)+1;   // установить длину сообщения
            cond_signal(&condx);         /* дать потоку записи сигнал
                                            для чтения сообщения */
        }
        else thr_yield();
        mutex_unlock(&mutx); // снять взаимоисключающую блокировку
    } while (1);

    FINISH();                            // завершение и выход
}

/* главный поток для управления потоком чтения и потоком записи */
main() {
    thread_t wtid, rtid, tid;

/* инициализировать взаимоисключающую блокировку и условную
   переменную */
    mutex_init(&mutx, USYNC_PROCESS, 0);
    cond_init(&condx, USYNC_PROCESS, 0);
    /* создать поток записи */
    if (thr_create(0,0,writer,0,0,&wtid))
        perror("thr_create");

    /* создать поток чтения */
    if (thr_create(0,0,reader,0,0,&rtid))
        perror("thr_create");
```

```
    /* ожидать завершения потока чтения */
    if (!thr_join(rtid,&tid,0))
    {
        done = 1;
        cond_signal(&condx);
    }

    /* удаление блокировки */
    mutex_destroy(&mutx);
    cond_destroy(&condx);
    thr_exit(0);
    return 0;
}
```

Функция *main* инициализирует взаимоисключающую блокировку *mutx* и условную переменную *condx*. Затем она создает поток записи и поток чтения для выполнения функций *writer* и *reader*. После этого главный поток ждет, пока поток чтения завершится с помощью функции *thr_join*, и сигнализирует потоку записи о необходимости завершения посредством глобальной переменной *done*. По завершении потоков чтения и записи главный поток удаляет взаимоисключающую блокировку и условную переменную (функциями *mutex_destroy* и *cond_destroy* соответственно).

Поток чтения читает одну или несколько строк, введенных пользователем со стандартного устройства ввода. Для каждой прочитанной строки он сначала устанавливает взаимоисключающую блокировку, чтобы иметь возможность доступа к глобальным переменным *msgbuf* и *msglen*. Если блокировку удается установить, поток помещает пользовательское сообщение в массив *msgbuf* и присваивает переменной *msglen* значение, равное длине текста сообщения. Затем с помощью функций *cond_signal* и *mutex_unlock* поток передает записывающему процессу сигнал о том, что нужно проверить переменную *msglen*. Поток чтения завершается, если не может прочитать введенный пользователем текст. В этом случае он снимает взаимоисключающую блокировку и самозавершается с помощью функции *thr_exit*.

Поток записи непрерывно опрашивает глобальную переменную *msglen*. Если ее значение не равно нулю, значит в массиве *msgbuf* есть сообщение, которое можно направить на стандартное устройство вывода. Для обработки каждого сообщения поток предварительно устанавливает взаимоисключающую блокировку, чтобы обеспечить себе исключительный доступ к переменным *msglen* и *msgbuf*. В случае успеха поток записи блокирует вызов функции *cond_wait* до тех пор, пока поток чтения или главный поток не вызовет функцию *cond_signal* для разблокирования этого вызова. После разблокирования поток записи проверяет, имеет ли переменная *done* ненулевое значение. Если имеет, то главный поток посылает в поток записи сигнал выхода. Если переменная *done* равна нулю, а значение переменной *msglen* не равно нулю, поток направляет сообщение, содержащееся в *msgbuf*, на стандартное устройство вывода. В противном случае он возвращается в цикл вызова *cond_wait* и ожидает поступления сообщения. Когда поток записи блокируется в вызове *cond_wait*, взаимоисключающая блокировка *mutx* снимается,

чтобы ее мог установить поток чтения или главный поток. Когда поток чтения или главный поток вызывает для переменной *condx* функцию *cond_signal*, функция *cond_wait*, прежде чем разблокировать поток записи, автоматически устанавливает взаимоисключающую блокировку.

Ниже приведен пример запуска программы *pipe.C* и полученные результаты:

```
% CC pipe.C -lthread -o pipe
% pipe
Have a good day
*>Have a good day
 Bye-Bye
*>Bye-Bye
5 exits
4 exits
```

13.5.2.3. Условные переменные стандарта POSIX.1c

Ниже приведены API POSIX.1c для управления условными переменными и соответствующие API Sun.

API POSIX.1c	API Sun
pthread_cond_init	cond_init
pthread_cond_wait	cond_wait
pthread_cond_timedwait	cond_timedwait
pthread_cond_signal	cond_signal
pthread_cond_broadcast	cond_broadcast
pthread_cond_destroy	cond_destroy

Прототипы этих функций выглядят следующим образом:

```
#include <pthread.h>

int pthread_cand_init (pthread_cond_t* condp, pthread_condatt_t* attr);
int pthread_cand_wait (pthread_cond_t* condp, pthread_mutex_t* mutxp);
int pthread_cand_timedwait (pthread_cond_t* condp,
                    pthread_mutex_t* mutxp, struct timespec* timp);
int pthread_cand_signal (pthread_cond_t* condp);
int pthread_cand_braadcast (pthread_cond_t* condp);
int pthread_cand_destray (pthread_cond_t* condp);
```

Значение аргумента *condp* представляет собой адрес переменной типа *pthread_cond_t*. Эта переменная является ссылкой на выделенную условную переменную. Аргумент *attr* функции *pthread_cond_init* — указатель на объект, содержащий атрибуты, которые задают свойства условной переменной. Если эта переменная должна иметь свойства по умолчанию, значение аргумента *attr* равно нулю.

По стандарту POSIX.1c условная переменная может инициализироваться статическим инициализатором PTHREAD_COND_INITIALIZER. Следующий код

```
pthread_cond_t    cond_var;
(void) pthread_cond_init(&cond_var, 0);
```

идентичен оператору

```
pthread_cond_t    cond_var = PTHREAD_COND_INITIALIZER;
```

API *pthread_cond_wait*, *pthread_cond_timedwait*, *pthread_cond_signal*, *pthread_cond_broadcast* и *pthread_cond_destroy* имеют то же назначение, что и API Sun *cond_wait*, *cond_timedwait*, *cond_signal*, *cond_broadcast* и *cond_destroy*. Описание этих API приведено в разделе 13.5.2.1.

13.5.3. Блокировки чтения-записи фирмы Sun

Блокировки чтения-записи подобны взаимоисключающим блокировкам, но могут устанавливаться как только для чтения, так и только для записи. Один или несколько потоков могут одновременно устанавливать блокировку чтения. Поток, который желает установить блокировку записи, будет заблокирован до тех пор, пока не будут сняты все блокировки чтения. Если поток устанавливает блокировку записи, ни один другой поток не может установить блокировку чтения или блокировку записи на данную блокировку до тех пор, пока предыдущий поток не снимет свою блокировку записи. Если два потока одновременно пытаются установить блокировку — один для чтения, другой для записи,— то предоставляется блокировка записи. Блокировки чтения-записи не так эффективны, как взаимоисключающие блокировки, но позволяют обеспечить одновременный доступ к данным для чтения нескольким процессам.

В стандарте POSIX.1c блокировки чтения-записи не определены; они присутствуют только в системах фирмы Sun. Ниже приведены многопотоковые библиотечные функции Sun для управления блокировками чтения-записи.

Функция	Назначение
rw_init	Инициализирует блокировку чтения-записи
rw_rdlock	Устанавливает блокировку чтения
rw_tryrdlock	Устанавливает блокировку чтения (вызывающий поток не блокируется)
rw_wrlock	Устанавливает блокировку записи
rw_trywrlock	Устанавливает блокировку записи (вызывающий поток не блокируется)
rw_unlock	Снимает блокировку чтения-записи
rw_destroy	Удаляет блокировку чтения-записи

Прототипы этих функций выглядят следующим образом:

```
#include <thread.h>

int rw_init (rwlock_t* rwp, int type, void* argp);
int rw_rdlock (rwlock_t* rwp);
int rw_tryrdlock (rwlock_t* rwp);
int rw_wrlock (rwlock_t* rwp);
int rw_trywrlock (rwlock_t* rwp);
int rw_unlock (rwlock_t* rwp);
int rw_destroy (rwlock_t* rwp);
```

Значение аргумента *rwp* представляет собой адрес переменной типа *rwlock_t*. Эта переменная является ссылкой на блокировку чтения-записи и устанавливается посредством функции *rw_init*. Аргумент *type* функции *rw_init* показывает, доступна ли данная блокировка чтения-записи для других процессов. Он может принимать следующие значения:

Значение аргумента *type*	Смысл
USYNC_PROCESS	Блокировка чтения-записи может использоваться потоками в других процессах
USYNC_THREAD	Блокировка чтения-записи может использоваться потоками только в вызывающем процессе

Аргумент *argp* функции *rw_init* в настоящее время не задействуется. Его значение должно быть равно 0.

Функция *rw_rdlock* пытается установить блокировку чтения, указанную аргументом *rwp*. Она блокирует вызывающий поток до успешного завершения операции.

Функция *rw_tryrdlock* аналогична функции *rw_rdlock*, но она неблокирующая. Если эта функция прерывается из-за того, что затребованная блокировка уже установлена как блокировка записи другим потоком, то возвращается ненулевое значение, а *errno* устанавливается в значение EBUSY.

Функция *rw_wrlock* пытается захватить блокировку записи, указанную аргументом *rwp*. Она блокирует вызывающий поток до успешного завершения операции.

Функция *rw_trywrlock* аналогична функции *rw_wrlock*, но она неблокирующая. Если эта функция прерывается из-за того, что затребованная блокировка принадлежит другому потоку, то возвращается ненулевое значение, а *errno* устанавливается в значение EBUSY.

Функция *rw_unlock* снимает блокировку чтения-записи, заданную аргументом *rwp*, а функция *rw_destroy* уничтожает ее.

Приведенная ниже программа *pipe2.C* — это новая версия программы *pipe.C* из предыдущего раздела. Для синхронизации потоков чтения и записи

в новой программе вместо взаимоисключающей блокировки и условной переменной использована блокировка чтения-записи.

```
#include <iostream.h>
#include <thread.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>

rwlock_t rwlk;
int     msglen, done = 0;
char    msgbuf[256];

void* writer (void* argp )
{
    while (!done) {
        if (rw_rdlock(&rwlk)) perror("rw_rdlock");  /* установить
                                                        блокировку
                                                        чтения */
        if (msglen) {           // проверить завершение сообщения
            cout << "*> " << msgbuf << endl; // вывести сообщение
            msglen = 0;
        }
        if (rw_unlock(&rwlk)) perror("rw_unlock(1)"); /* снять
                                                        блокировку
                                                        чтения-
                                                        записи */
        thr_yield();            /* передать другому потоку */
    }
    /* завершить поток */
    cerr << "write thread (" << (int)thr_self() << ") exits\n";
    thr_exit(0);
    return 0;
}

void* reader (void* argp )
{
    do {
        if (rw_wrlock(&rwlk)) perror("rw_wrlock"); /* установить
                                                      блокировку
                                                      записи */
        if (!msglen) {                            // буфер пуст?
            if (!cin.getline(msgbuf,256)) break; /* получить
                                                    сообщение от
                                                    пользователя */
            msglen = strlen(msgbuf)+1; // установить размер сообщения
        }
        if (rw_unlock(&rwlk)) perror("rw_unlock(2)"); /* снять
                                                        блокировку
                                                        чтения-
                                                        записи */
        thr_yield();                        //передать другому потоку
```



```
    } while (1);
    /* завершить поток */
    cerr << "read thread (" << (int)thr_self() << ") exits\n";
    if (rw_unlock(&rwlk)) perror("rw_unlock(3)");
    thr_exit(0);
    return 0;
}

main() {
    thread_t wtid, rtid, tid;

    if (rwlock_init(&rwlk, USYNC_PROCESS, 0)) {
        perror("rwlock_init");
        return 1;
    }

    /* создать поток записи */
    if (thr_create(0,0,writer,0,0,&wtid))
        perror("thr_create");

    /* создать поток чтения */
    if (thr_create(0,0,reader,0,0,&rtid))
        perror("thr_create");

    /* ожидать завершения процесса чтения */
    if (!thr_join(rtid,&tid,0))
    {
        done = 1;
    }

    /* очистка */
    rwlock_destroy(&rwlk);

    thr_exit(0);
}
```

Функция *main* инициализирует блокировку чтения-записи, а затем создает потоки чтения и записи для совместной работы. Главный поток ждет завершения потока чтения, а затем через переменную *done* посылает в поток записи сигнал завершения. Затем главный поток удаляет блокировку чтения-записи и завершается.

Поток чтения выполняет функцию *read*. Она получает от пользователя одну или несколько строк текста и для каждой введенной строки устанавливает блокировку чтения-записи для доступа с целью записи, обеспечивая потоку исключительное право доступа к переменным *msgbuf* и *msglen*. После установки блокировки записи поток записывает текст сообщения в массив *msgbuf*, записывает в *msglen* размер текста сообщения и снимает блокировку чтения-записи. Затем данный поток передает выполнение потоку записи, чтобы последний получил доступ к блокировке и переменной *msgbuf*. Затем поток берет следующее пользовательское сообщение и повторяет всю

процедуру. Когда поток не может прочитать пользовательский текст, он снимает блокировку чтения-записи и завершается с помощью функции *thr_exit*.

Поток записи выполняет функцию *write*. Она читает сообщение из потока чтения и посылает его на стандартное устройство вывода. Для каждого сообщения функция сначала устанавливает блокировку чтения и следит за тем, чтобы значение переменной *msglen* было больше нуля. Если в переменной *msgbuf* есть сообщение, поток подает его на стандартное устройство вывода и сбрасывает значение переменной *msglen* в 0. После этого поток снимает блокировку чтения-записи и передает выполнение потоку чтения, чтобы последний получил доступ к блокировке и поместил следующее сообщение в буфер *msgbuf*. Если главный поток устанавливает глобальную переменную *done* в ненулевое значение, значит, сообщений больше нет, и поток записи завершается с помощью функции *thr_exit*.

Ниже приведен пример запуска программы *pipe2.C* и полученные результаты:

```
% CC pipe2.C -lthread -o pipe2
% pipe2
Have a good day
*>Have a good day
Bye-Bye
*>Bye-Bye
^D
read thread (5) exits
write thread (4) exits
%
```

13.5.4. Семафоры

И в библиотеке многопотоковых функций Sun, и в стандарте POSIX.1c имеются API для управления семафорами. Эти построенные на потоках семафоры похожи на семафоры UNIX System V тем, что каждый семафор имеет целочисленное значение, которое должно быть больше или равно 0. Значение семафора может устанавливаться любым потоком, который имеет к нему доступ. Если поток пытается уменьшить значение семафора так, что в результате получится отрицательное число, этот поток блокируется. Он остается в заблокированном состоянии до тех пор, пока другой поток не увеличит значение семафора до нулевого или положительного значения.

Блокирующие свойства семафоров можно использовать для синхронизации выполнения определенных сегментов кода и для доступа нескольких потоков к совместно используемым данным. Прежде чем обратиться к разделяемому ресурсу, каждый поток пытается уменьшить значение семафора на единицу. Успеха достигает лишь один из них, а остальные блокируются. После того как этот поток закончил использовать разделяемый ресурс, он увеличивает значение семафора до предыдущего уровня, чтобы другой поток мог разблокироваться и продолжать работу с ресурсом.

Семафоры можно использовать вместо взаимоисключающих блокировок и условных переменных. Однако следует отметить, что взаимоисключающие блокировки могут сниматься только потоками, которые ими владеют, тогда как изменять значение семафора может любой поток, имеющий к нему доступ. Обеспечение надежности программ требует дополнительных усилий при программировании, поскольку приходится следить за тем, что именно каждый поток делает с семафором.

POSIX.1c использует те же самые API семафоров, которые в POSIX.1b применяются для синхронизации потоков. Описание этих API вы найдете в разделе 10.6. Sun Microsystems использует для синхронизации потоков другой набор API семафоров. В приведенной ниже таблице показаны эти API и соответствующие API семафоров POSIX.1b:

API Sun	API POSIX.1b	Назначение
sema_init	sem_init	Инициализирует семафор
sema_post	sem_post	Увеличивает значение семафора на 1
sema_wait	sem_wait	Уменьшает значение семафора на 1
sema_trywait	sem_trywait	Уменьшает значение семафора на 1 без блокировки потока
sema_destroy	dem_destroy	Уничтожает семафор

API семафоров Sun имеют следующие прототипы:

```
#include <synch.h>

int sema_init (sema_t* svp, int init_val, init type, void* argp);
int sema_wait (sema_t* svp);
int sema_trywait (sema_t* svp);
int sema_post (sema_t* svp);
int sema_destroy (sema_t* svp);
```

Значение аргумента *svp* представляет собой адрес переменной типа *sema_t*. Эта переменная устанавливается как ссылка на семафор посредством функции *sema_init*. Аргумент *type* функции *sema_init* показывает, доступен ли данный семафор для других процессов. Он может принимать следующие значения.

Значение аргумента *type*	Смысл
USYNC_PROCESS	Семафор может использоваться потоками в других процессах
USYNC_THREAD	Семафор может использоваться потоками только в вызывающем процессе

Аргумент *argp* функции *sema_init* в настоящее время не задействуется. Его значение должно быть равно 0.

Функция *sema_wait* пытается уменьшить значение семафора, указанное в аргументе *svp*. Эта функция блокирует вызывающий поток до тех пор, пока операция не будет завершена успешно.

Функция *sema_trywait* похожа на функцию *sema_wait*, но она неблокирующая. Если эта функция прерывается из-за того, что значение затребованного семафора не может быть уменьшено, она возвращает ненулевое значение и устанавливает *errno* в EBUSY.

Функция *sema_post* увеличивает значение семафора, заданного аргументом *svp*.

Функция *sema_destroy* уничтожает семафор, указанный аргументом *svp*.

Приведенная ниже программа *pipe3.C* — это новая версия программы, представленной в предыдущем разделе. Для синхронизации потоков чтения и записи в новой программе вместо блокировки чтения-записи используется семафор.

```
#include <iostream.h>
#include <thread.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>

#define FINISH()   { cerr << {int}thr_self() << " exits\n"; \
                     thr_exit(0); return 0; }

sema_t  semx;
int     msglen, done;
char    msgbuf[256];

void* writer (void* argp )
{
    do {
        sema_wait(&semx);    // захватить семафор
        if (msglen) {        // проверить наличие сообщения в буфере
            cout << "*> " << msgbuf << endl; // напечатать сообщение
            msglen = 0;      // сбросить размер буфера сообщений
        }
        sema_post(&semx);    // освободить семафор
        thr_yield();         // позволить выполняться другим потокам
    } while (!done);         // выполнять, пока не закончатся сообщения
    FINISH();                // завершить поток
}

void* reader (void* argp )
{
    do {
        sema_wait(&semx);      // захватить семафор
        if (!msglen) {         // проверить, пуст ли буфер
            if (cin.getline(msgbuf,256)) // получить новое сообщение
                msglen = strlen(msgbuf)+1; /* установить размер
                                              сообщения в msglen */
```



```
            else done = 1; //входных сообщений больше нет
        }
        sema_post(&semx);   // освободить семафор
        thr_yield();        // позволить выполняться другим потокам
    } while (!done);        // выполнять, пока не закончатся сообщения
    FINISH();               // завершить поток
}

main() {
    thread_t wtid, rtid, tid;
    /* инициализировать семафор начальным значением 1 */
    sema_init(&semx, 1, USYNC_PROCESS, 0);

    /* создать поток записи */
    if (thr_create(0,0,writer,0,0,&wtid))
        perror("thr_create");

    /* создать поток чтения */
    if (thr_create(0,0,reader,0,0,&rtid))
        perror("thr_create");

    /* ожидать завершения всех потоков */
    while (!thr_join(0,0,0)) ;

    /* очистка */
    sema_destroy(&semx);
    thr_exit(0);
    return 0;
}
```

Эта программа аналогична двум предыдущим и отличается только тем, что вместо взаимоисключающей блокировки, условной переменной и блокировки чтения-записи используется семафор. Этот семафор, *semx*, инициализируется в главном потоке начальным значением 1. При запуске и поток чтения, и поток записи пытаются захватить этот семафор вызовом функции *sema_wait*, но успеха может достичь только один.

Если поток записи захватывает семафор первым, он обнаруживает, что буфер сообщений пуст, и освобождает семафор, передавая право выполнения потоку чтения. Захватив семафор, поток чтения читает сообщение пользователя и помещает его в массив *msgbuf*. Затем он присваивает переменной *msglen* значение, равное размеру текста сообщения. После этого поток чтения освобождает семафор и передает право выполнения потоку записи.

Разблокированный поток записи обнаруживает, что буфер *msgbuf* не пуст, и направляет содержащееся в нем сообщение на стандартное устройство вывода. Затем он сбрасывает переменную *msglen* и освобождает семафор. Начинается следующий цикл обработки сообщений потоком чтения и потоком записи.

Если поток чтения не может прочитать сообщение (возможно, из-за признака конца файла), он устанавливает переменную *done* в 1, сообщая

таким образом, что и поток записи, и поток чтения должны завершиться посредством вызова функции *thr_exit*.

Ниже приведен пример запуска программы ***pipe3.C*** и показаны полученные результаты:

```
% CC pipe3.C -lthread -o pipe3
% pipe3
Have a good day
*>Have a good day
Bye-Bye
*>Bye-Bye
^D
5 exits
4 exits
%
```

13.6. Данные потоков

Переменными и памятью, динамически выделяемой в функции, владеют все потоки, которые эту функцию выполняют. Глобальные и статические переменные могут совместно использоваться множеством потоков, но для синхронизации доступа к таким общим данным потоки должны применять взаимоисключающие блокировки, условные переменные и т.д. Однако некоторые глобальные переменные нельзя синхронизировать. Например, переменная *errno*, определенная в библиотеке С, устанавливается каждым системным вызовом. Если несколько потоков одновременно выполняют системные вызовы, то переменная *errno* для разных потоков должна устанавливаться в разные значения. Эту проблему можно преодолеть, потребовав, чтобы только один поток выполнял системный вызов в каждый момент времени. В результате многопотоковые программы будут работать в однопотоковом режиме.

Проблема с переменной *errno* разрешается библиотекой потоков, которая автоматически создает частную копию *errno* для каждого потока, выполняющего системные вызовы. Таким образом, несколько потоков могут одновременно выполнять системные вызовы и проверять собственные значения переменной *errno*.

Подобная проблема возникает и в ситуации, когда функции содержат статические переменные, к которым одновременно обращаются несколько потоков. Например, библиотечная С-функция *ctime* возвращает символьную строку местного времени и даты. Эта строка хранится во внутреннем статическом буфере функции *ctime*:

```
const char* ctime (const time_t *timval)
{
    static char timbuf[...];
    /* конвертировать timval в местную дату и время и занести
       результат в timbuf */
    return timbuf;
}
```


Таким образом, если несколько потоков вызывают эту функцию одновременно, она должна суметь возвратить разным потокам разные результаты. Эту проблему можно было бы разрешить, динамически выделяя буферы для хранения затребованной даты и времени в каждом вызове, однако такой подход сопряжен с рядом трудностей, а именно:

- время выполнения функции и объем необходимой для нее памяти увеличиваются; в результате производительность программ, использующих эту функцию, падает, а требования к памяти возрастают;
- существующие программы, которые используют эту функцию, необходимо исправить так, чтобы динамически выделяемая память, занимаемая этой функцией, освобождалась;
- неисправленные однопотоковые программы нельзя использовать в многопотоковой среде.

Чтобы преодолеть эти трудности, POSIX.1c и Sun Microsystems определяют реентерабельные версии популярных библиотечных функций. Эти новые функции могут одновременно вызываться множеством потоков, при этом побочные эффекты отсутствуют, а производительность остается такой же, как у однопотоковых версий, и даже может повыситься. Имена этих реентерабельных функций отличаются от имен аналогичных однопотоковых версий суффиксом *_r*. Так, реентерабельная версия функции *ctime* называется *ctime_r*. Все эти новые функции принимают один дополнительный аргумент — адрес переменной, содержащей возвращаемое значение. Эта переменная определяется вызывающими потоками. Таким образом, несколько потоков могут одновременно вызвать одну и ту же функцию, и каждый из них получит ответ через соответствующую переменную. Однопотоковые и существующие многопотоковые программы могут продолжать пользоваться старыми библиотечными функциями, и многопотоковая среда их не касается. Новые многопотоковые программы должны использовать реентерабельные версии библиотечных функций.

Новая функция *ctime_r* определяется следующим образом:

```
char* ctime_r (const time_t* timval, char buf[])
{
    /* конвертировать timval в местную дату и время и занести
       результат в buf */
    return buf;
}
```

Отметим, что функция *ctime_r* использует свой входной аргумент *buf* для хранения затребованных вызывающим потоком значений даты и времени и возвращает этому потоку адрес *buf*.

Фирма Sun Microsystems создала реентерабельные версии библиотечных функций С, библиотечных функций работы с гнездами и математических библиотечных функций.

Даже при наличии реентерабельных функций и динамически определяемых для каждого потока глобальных переменных, поддерживаемых библиотекой

потоков, все равно может возникнуть необходимость в пользовательских данных для конкретных потоков. Так, пользователи, разрабатывающие пакет утилит (например, новый пакет GUI), который могут использовать другие программисты, возможно, захотят определить собственные версии *errno*, чтобы иметь возможность проверить, не содержит ли эта переменная кода ошибки, возвращенной утилитами. При этом, однако, функции пакета могут одновременно вызываться несколькими потоками, поэтому необходимо, чтобы переменная *errno* определялась функциями конкретно для каждого потока. Ниже мы покажем, как это делается, а сейчас перечислим функции, определенные в библиотеке потоков Sun для управления данными потоков.

Функция	Назначение
thr_keycreate	Определяет общую переменную ("ключ") для всех потоков
thr_setspecific	Заносит данные потока в "ключ"
thr_getspecific	Извлекает данные потока из "ключа"

Прототипы этих функций выглядят следующим образом:

```
#include <thread.h>

int thr_keycreate (thread_key_t* keyp, void (*destr)(void*));
int thr_setspecific (thread_key_t key, void* valuep);
int thr_getspecific (thread_key_t key, void* valuep);
```

Значение аргумента *keyp* представляет собой адрес переменной типа *thread_key_t*. Эта переменная инициализируется функцией *thr_keycreate*. Необязательный аргумент *destr* — это адрес определяемой пользователем функции, которая может вызываться для уничтожения данных после завершения потока, зарегистрировавшего эти данные в переменной **keyp*. Значение этого аргумента, передаваемое в функцию *destr*, представляет собой адрес данных завершающегося потока, зарегистрированных в **keyp* с помощью функции *thr_setspecific*.

Функция *thr_setspecific* вызывается потоком для регистрации данных в "ключе". Аргумент *key* содержит "ключ", в котором необходимо зарегистрировать данные. Аргумент *valuep* содержит адрес данных потока. "Ключ" может одновременно содержать несколько значений, но только по одному значению на поток.

Для выборки данных вызывающего потока, зарегистрированных в "ключе", обозначенном аргументом *key*, вызывается функция *thr_getspecific*. Данные потока возвращается через аргумент *valuep*.

В заголовке файла *pkg.h* определяется класс данных, который может одновременно использоваться несколькими потоками. В частности, этот класс определяет объект *errno* и объект *ofstream* для каждого потока, поэтому

никаких конфликтов между потоками, вызывающими одни и те же функции класса, не возникает. Приведем содержимое файла *pkg.h*:

```
#ifndef PKG_H
#define PKG_H
#include <fstream.h>
#include <stdio.h>
#include <thread.h>

/* запись для хранения набора данных конкретного потока */
class thr_data {
        int      errno;        // errno для потока выполнения
        ofstream& ofs;         // поток вывода для потока выполнения
        /* прочее */
    public:
        /* функция-конструктор */
        thr_data( int errval, ofstream& os)
            : errno(errval), ofs(os)
                    {};
        /* функция-деструктор */
        ~thr_data()   ( ofs.close();  };

        /* возвращает значение errno для потока */
        int& errval() { return errno; };

        /* возвратить указатель на поток вывода для потока
           выполнения */
        ofstream& os()( return ofs;   );

        /* другие функции-члены */
};

/* Класс пакета утилит */
class Xpackage {
        thread_key_t   key;    // ключ для всех потоков
        ofstream       ocerr;  // выходной поток по умолчанию
        /* другие данные */
    public:
        /* вызывается, когда поток уничтожается. Данные потока
           удаляются */
        friend void destr( void* valueP )
        {
              thr_data  *pDat = (thr_data*)valueP;
        delete pDat;
        };

        /* функция-конструктор */
        Xpackage() : ocerr("err.log")
        {
              if (thr_keycreate(&key,destr))
                    perror("thr_create");
        };
```

```
/* функция-деструктор */
~Xpackage()   ( ocerr.close(); };

/* вызывается, когда поток открывается */
void new_thread( int errval,  ofstream& os )
{
      thr_data        *pDat;
      /* распределение памяти для данных потока */
      pDat = new thr_data(errval, os);
      if (thr_setspecific(key,pDat))
            perror( "thr_setspecific");
};

/* устанавливает errno потока и возвращаемое значение */
int set_errno( int rc )
{
      thr_data        *pDat;
      if (thr_getspecific(key,(void**)&pDat))
            perror("thr_getspecific");
      else pDat->errval() = rc;
      return rc==0 ? 0 : -1;
};

/* возвращает текущее значение errno для потока */
int errno()
{
      thr_data        *pDat;
      if (!thr_getspecific(key,(void**)&pDat))
      {
            return pDat->errval();
      }
      perror("thr_getspecific");
      return -1;
};

/* возвращает указатель на поток вывода данного потока
   выполнения */
ofstream& os()
{
      thr_data        *pDat;
      if (!thr_getspecific(key,(void**)&pDat))
      {
            return pDat->os();
      }
      perror("thr_getspecific");
      return ocerr;
};

/* пример функции из пакета */
int chgErrno(int new_val )
{
      new_val += (int)thr_self();
```



```
            return set_errno( new_val );
      };

      /* другие функции из пакета */
};

#endif
```

Все данные потоков хранятся в записи типа *thr_data*. В этой записи содержатся объекты *errno* и *ofstream*, которые в каждом потоке являются закрытыми (private). При желании пользователи могут переопределить класс *thr_data* так, чтобы он содержал дополнительные данные.

В каждой пользовательской программе должна присутствовать глобальная переменная типа *Xpackage*. При запуске программы вызывается конструктор *Xpackage::Xpackage*, который инициализирует переменную *Xpackage::key*, совместно используемую всеми потоками процесса. Переменная *Xpackage::ocerr* обозначает поток вывода по умолчанию в случае, если определенный в потоке выполнения поток вывода не может быть открыт. Если пользователь хочет сохранять в процессе данные потоков выполнения других типов, он может определить несколько переменных типа *Xpackage*, по одной для каждого типа данных.

Функции пакета вызывают *Xpackage::set_errno*, чтобы установить значение *errno* для вызывающего потока. Функция *Xpackage::errno* вызывается потоком для получения конкретного значения переменной *errno*. Функция *Xpackage::os* возвращает указатель на поток вывода для данного потока выполнения.

Функция *Xpackage::chgErrno* в приведенном примере только устанавливает значение *errno* для каждого потока равным сумме идентификатора потока и числа 100. Как и все остальные определенные в пакете функции, в случае успешного выполнения она возвращает 0, а в случае неудачи — -1 (в *errno* конкретного потока соответствующим образом устанавливается код ошибки).

Функция *destr* вызывается каждый раз, когда поток завершается. Эта функция удаляет данные потока типа *thr_data*. Если поток зарегистрировал несколько записей типа *thr_data* в нескольких переменных типа *Xpackage*, то функция *destr* вызывается несколько раз, т.е. для каждого элемента данных типа *thr_data*, принадлежащего завершающемуся потоку.

Приведенный ниже файл *thr_errno.C* — это пример пользовательской программы, в которой используется класс *Xpackage*.

```
#include "pkg.h"

Xpackage pkgObj; /* объект пакета */

/* функция, выполняемая каждым потоком */
void* funcl( void* argp )
{
    int *rcp = new int(1);
```

```
    /* открыть поток вывода потока выполнения */
    ofstream ofs ((char*)argp);
    if (!ofs) thr_exit((void**)&rcp);

    /* инициализация данных потока */
    pkgObj.new_thread( 0, ofs );

    /* выполнить работу с помощью функций пакета */
    pkgObj.chgErrno( 100 );  /* изменить errno потока выполнения*/

    /* записать данные в поток вывода потока выполнения */
    pkgObj.os() << (char*)argp << " ["
                << (int)thr_self() << "] finishes\n";

    /* поток закрывается; установить код возврата */
    *rcp = pkgObj.errno();
    thr_exit(rcp);
    return 0;
}

/* главная функция потока */
int main(int argc, char** argv)
{
    thread_t tid;
    int      *rc;

    /* создать поток для каждого аргумента командной строки */
    while (--argc > 0)
        if (thr_create(0,0,func1,(void*)argv[argc],0,&tid))
            perror("thr_create");

    /* ждать завершения всех потоков */
    while (!thr_join(0,&tid,(void**)&rc))
    {
        cerr << "thread: " << (int)tid << " exists. rc=" << *rc << endl;
            delete rc;
    }

    /* завершить главный поток */
    thr_exit(0);
    return 0;
}
```

Эта программа вызывается с одним или несколькими именами файлов в качестве аргументов командной строки. Каждый аргумент — это имя файла, в который направлен вывод соответствующего потока выполнения. Для каждого аргумента главный поток создает новый поток, чтобы выполнить функцию *func1*. Аргумент функции *func1* — имя файла, в который направлен вывод нового потока выполнения. После создания всех потоков выполнения главный поток ждет их завершения, а потом завершается сам.

Начиная выполнять функцию *func1*, поток определяет объект *ofstream*, обозначающий заданный выходной файл. Затем он вызывает функцию *pkgObj.new_thread*, которая выделяет область памяти для хранения данных потока и в которой содержится объект *ofstream*. Значение *errno* устанавливается в нуль. После этого поток вызывает функции пакета для выполнения основных операций. Затем вызывается функция *pkgObj.chgErrno*, которая устанавливает значение *errno* потока равным сумме его идентификатора и числа 100. В результате значение *errno* каждого потока, который выполняет функцию *func1*, уникально. Затем поток вызывает функцию *pkjObj.os*, которая возвращает указатель на объект потока вывода и посылает туда имя выходного файла и идентификатор потока выполнения. Поток выполнения завершается, указывая свое значение *errno* как аргумент вызова функции *thr_exit*.

Ниже приведен пример выполнения программы *thr_errno* и полученные результаты:

```
% CC thr_errni.C -o thr_errno -lthread
% thr_errno  a  b
thread: 5 exits. rc=105
thread: 4 exits. rc=104
% cat a
a [5] finishes
% cat b
b [4] finishes
```

Здесь программа *thr_errno* вызывается с именами двух файлов: *a* и *b*. Для выполнения функций *func1* создаются два потока. Первый поток имеет идентификатор 5 и создает выходной файл с именем *a* и содержимым *a [5] finishes*. Этот поток завершается с кодом выхода 104. Идентификатор второго потока — 4. Он создает файл с именем *b* и содержимым *b [4] finishes*. Его код выхода — 104.

13.7. Среда многопотокового программирования

Для поддержки многопотокового программирования в ОС Solaris имеется библиотека потоков, которая позволяет пользователям создавать в своих программах потоки выполнения и управлять ими. Есть также модификации стандартных библиотек, например имеются реентерабельные версии многих популярных библиотечных функций. Глобальные переменные (например, *errno*), экспортируемые из стандартных библиотек, определяются динамически для каждого потока, который ими пользуется. Все это позволяет успешно использовать стандартные библиотеки одновременно в нескольких потоках.

Кроме того, в ОС Solaris модифицировано ядро, что обеспечивает поддержку симметричной многопроцессорной обработки и планирования облегченных процессов. Имеются также многопотоковые версии команд *debugger* и *truss*, которые отлаживают и трассируют выполнение отдельных

потоков в процессе. Ожидается, что такие возможности предоставят и другие фирмы, которые поддерживают в своих системах многопотоковые программные среды.

13.8. Пример распределенного многопотокового приложения

В этом разделе рассматривается многопотоковая распределенная программа. Это интерактивная программа, которая выполняет пользовательские команды shell на любой машине в локальной сети. Для связи с выделенными RPC-серверами, работающими на удаленных хост-машинах, в программе используются удаленные вызовы процедур. Когда пользователь вводит команду shell, программа создает поток для установления соединения с RPC-сервером, работающим на указанной пользователем хост-машине. Этот сервер выполняет пользовательскую команду на указанной машине, а поток проверяет его код завершения и в случае ошибки выставляет пользователю соответствующий флаг.

Используя отдельный поток для обработки каждой пользовательской команды, программа может принимать новую команду, продолжая выполнять одну или несколько предыдущих команд другими потоками. Таким образом, для ввода команды пользователям не приходится ждать, пока завершится выполнение предыдущей команды. Благодаря этому такая программа более "интерактивна", чем соответствующая однопотоковая версия. К тому же программа может свободно использовать все ресурсы, имеющиеся на многопроцессорной машине.

Используя RPC, программа может распределить рабочую нагрузку на несколько сетевых компьютеров, благодаря чему значительно повышается ее производительность и гибкость. Используя метод широковещательных RPC, программа автоматически определяет, на каких хост-машинах работают RPC-серверы. Таким образом, от пользователя требуется лишь загрузить RPC-серверы на выбранные им машины (это могут быть и гетерогенные платформы, например рабочие UNIX-станции, VMS-машины и машины с Windows NT) и запустить программу на хост-машине, которая поддерживает многопотоковые программы (например, на рабочей станции Sun с ОС Solaris).

RPC-сервер, который взаимодействует с этой интерактивной программой и выполняет пользовательские команды shell, содержится в программе *shell_svc.C*. В этой программе используются классы RPC, определенные в главе 12. Они позволяют создать объект *RPC_svc* для работы RPC. Ниже приведен файл *shell_svc.C*:

```
#include "mshell.h"
#include "RPC.h"

RPC_svc *svcp;              // указатель на объект RPC-сервера
```



```
/* RPC-функция для выполнения одной пользовательской команды */
int execshell( SVCXPRT* xtrp )
{
    static int   res=0, rc= RPC_SUCCESS;
    char  *cmd = 0;

    /* получить команду shell от RPC-клиента */
    if (svcp->getargs( xtrp, (xdrproc_t)xdr_string,
                       (caddr_t)&cmd)!=RPC_SUCCESS)
        return -1;

    /* выполнить команду с помощью функции system */
    res = system(cmd);

    /* послать результат выполнения RPC-клиенту */
    if (svcp->reply(xtrp, (xdrproc_t)xdr_int, (caddr_t)&res)!=RPC_SUCCESS)
        rc =  -2;

    return rc;
)

int main(int argc, char* argv[])
(
    /* создать объект RPC-сервера для функции execshell */
    RPC_svc *svcp = new RPC_svc( SHELLPROG, SHELLVER,
                                 argc==2 ? argv[1] : "netpath");
    if (!svcp || !svcp->good()) return 1;
    /* ждать запросы RPC-клиентов */
    if (svcp->run_func( EXECSHELL, execshell )) return 3;
    return 0; /* shouldn't get here */
)
```

Программа RPC-сервера создает объект *RPC_svc* для выполнения функции *execshell* по запросу RPC-клиента. Функция *execshell* принимает от клиента символьную строку, которая завершается символом NULL и содержит команду shell ОС UNIX. Для выполнения этой команды вызывается функция *system*. Значение, возвращенное вызовом *system*, передается RPC-клиенту (функцией *RPC_svc::reply*). Серверный RPC-процесс — это демон, который работает в фоновом режиме до тех пор, пока система не закроется или пока он не будет явно уничтожен пользователем.

Заголовок *RPC.h* и файл *RPC.C* рассмотрены в главе 12 (раздел 12.5). В следующем ниже заголовке *mshell.h* заданы номер программы, номер версии и номер процедуры для функции *execshell*. Кроме этого в заголовке определен класс shellObj, назначение которого поясняется ниже:

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#ifndef _SHELL_H_RPCGEN
#define _SHELL_H_RPCGEN
#include <iostream.h>
```

```
#include <rpc/rpc.h>
#include <thread.h>
#include <string.h>

#define    SHELLPROG ((unsigned long)(0x20000001))
#define    SHELLVER  ((unsigned long)(1))
#define    EXECSHELL ((unsigned long)(1))

typedef void* (*FNPTR)( void *);

class shellObj {
      char*   help_msg;
      FNPTR   action;
      void*   (*fnptr)( void* );
      int     create_thread;
   public:
      /* конструктор. Задает сообщение-подсказку, функцию action
         и значение errno */
      shellObj( const char* msg, FNPTR func, int thr_ok=1) {
         help_msg = new char[strlen(msg)+1];
         strcpy(help_msg,msg);
         action  = func;
         create_thread = thr_ok;
      };

      // функция-деструктор
      ~shellObj() { delete help_msg; };

      ostream& usage( ostream& os, int idx) {
            os << idx << ": " << help_msg << endl;
            return os;
      };

      // выполнить функцию action с помощью потока
      int doit( void* argp ) {
            if (create_thread) {
                  thread_t  tid;
                  if (thr_create(0,0,action,argp,THR_DETACHED,&tid))
    perror("thr_create");
                  return (int)tid;
            }
            else return (int)action(argp);
      };
};

#endif /* !_SHELL_H_RPCGEN */
```

Программа RPC-сервера компилируется и запускается на любом компьютере, где поддерживаются RPC-функции стиля UNIX System V.4:

```
% CC -DSYSV4 shell_svc.C RPC.C -o shell_svc -lnsl
% shell_svc &
```

Главная программа — *main_shell.C*. Это управляемая в режиме меню программа, которая многократно отображает на устройстве стандартного

вывода меню вариантов. Пользователь выбирает вариант, используя индексы меню, и программа выполняет этот вариант в отдельном потоке. Программа *main_shell.C* приведена ниже:

```
#include <strstream.h>
#include <thread.h>
#include <string.h>
#include <stdio.h>
#include "mshell.h"

/* определить удаленные хост-машины, на которых выполняются
   серверные процессы */
extern void* collect_hosts( void* argp );
extern void* display_hosts( void* argp );
extern void* exec_shell( void* argp );

shellObj *funcList[4];                    // диспетчерская таблица
int    numFunc = sizeof(funcList)/sizeof(funcList[0]);
rwlock_t    rwlck;

/* выход из программы */
void* quit_prog( void* argp )
{
    return (void*)0;
}

/* выполнить на хост-машине одну команду shell */
void* getcmd( void* argp )
{
    char host[20], cmd[256], cmd2[256];
    cout << "shell cmd> " << flush;
    cin >> cmd;
    cin.getline(cmd2,256);
    cout << "host: " << flush;
    cin >> host;
    if (!cin)
        cout << "Inavlid input\n" << flush;
    else {
        if (strlen(cmd2)) strcat(strcat(cmd," "),cmd2);
        strcat(strcat(cmd,"/"),host);
        char* ptr = new char[strlen(cmd)+1];
        ostrstream(ptr,strlen(cmd)+1) << cmd;
        if (thr_create(0,0,exec_shell,ptr,THR_DETACHED,0))
            perror("thr_create");
    }
    return (void*)1;
}

/* прочитать выбранный пользователем вариант и выполнить его */
int exec_cmd()
{
    char buf[256];
    cout << "Selection> " << flush;
    cin >> buf;
```

```
    cout << endl;
    if (cin) {
        int  idx = -1;
        istrstream(buf) >> idx;
        if (idx >=0 && idx < numFunc)
            return funcList[idx]->doit(0);
        else cerr << "Invalid input\n";
    }
    return 0;
}

/* вывести меню */
void display_menu(ostream& os)
{
    for (int i=0; i < numFunc; i++)
        (void)funcList[i]->usage(os,i);
}

/* инициализация блокировки чтения-записи и диспетчерской таблицы */
int init_all()
{
    if (rwlock_init (&rwlck, USYNC_THREAD, 0)) {
        perror("rwlock_init");
        return -1;
    }
    funcList[0] =  new shellObj( "Collect host names", collect_hosts);
    funcList[1] =  new shellObj( "Display host names", display_hosts);
    funcList[2] =  new shellObj( "Execute a shell command", getcmd, 0);
    funcList[3] =  new shellObj( "Quit", quit_prog, 0);
    return 0;
}

/* главный модуль */
int main() {
    if (init_all() == 0)
        do {
            display_menu(cerr);
            if (!exec_cmd()) break;
        } while (1);
    thr_exit(0);
    return 0;
}
```

В этой интерактивной программе пользователю предлагается меню из четырех пунктов:

Индекс	Функция	Назначение
0	collect_hosts	Находит все хост-машины, на которых работает RPC-сервер
1	display_hosts	Выводит имена всех хост-машин, на которых работает RPC-сервер
2	getcmd	Выполняет команду shell
3	quit_prog	Выход из интерактивной программы

Адреса этих функций хранятся в диспетчерской таблице *funcList*. Когда пользователь вводит номер варианта, это целое значение используется как индекс в диспетчерской таблице для вызова соответствующей функции. Например, если пользователь введет 0, то будет вызвана функция *collect_hosts*. Каждый элемент таблицы *funcList* — это указатель на объект *shellObj*. Класс *shellObj* определяется в заголовке *mshell.h*. Каждый объект *shellObj* содержит справочное сообщение, поясняющее его использование, указатель на определяемую пользователем функцию (которая может вызываться для выполнения фактической задачи объекта) и флага, который показывает, нужно ли вызывать эту функцию через поток. Создаются четыре объекта *shellObj* для четырех функций — *collect_hosts*, *display_hosts*, *getcmd*, *quit_prog*. Они обозначаются с использованием элементов 0, 1, 2 и 3 таблицы *funcList*.

Функция *main* нашей интерактивной программы сначала вызывает функцию *init_all*, которая инициализирует блокировку чтения-записи *rwlck* и диспетчерскую таблицу *funcList* с четырьмя объектами *shellObj*. После этого главная функция входит в цикл, вызывая функцию *display_menu* для вывода меню на экран. Затем вызывается функция *exec_cmd*, которая приглашает пользователя выбрать элемент меню и вызывает функцию, соответствующую этому элементу. Цикл завершается, когда *exec_cmd* возвращает нулевое значение, и функция *main* завершается через функцию *thr_exit*.

Функция *display_menu* просматривает диспетчерскую таблицу *funcList* и вызывает функцию *shellObj::usage* для каждого объекта *shellObj*, обозначенного элементом таблицы. В результате каждый объект *shellObj* направляет в поток вывода *cerr* информацию о своем использовании и соответствующий ему индекс диспетчерской таблицы.

Вариант, выбранный пользователем в меню, читается функцией *exec _cmd*. Эта функция следит за тем, чтобы индекс выбранного пользователем элемента лежал в диапазоне 0-3; в противном случае она выставляет флаг ошибки. Если пользователь выбрал допустимый вариант, функция *exec_cmd* использует эту цифру как индекс таблицы *funcList* для получения указателя на объект *shellObj*. Затем она вызывает функцию *shellObj::doit* этого объекта. Функция *exec_cmd* возвращает нулевое значение, если не может получить данные от пользователя, либо значение, возвращенное функцией *shellObj::doit*.

Функция *exec_cmd* передает каждой вызываемой ею функции *shellObj::doit* нулевое значение аргумента. Для элементов меню 0 и 1 соответствующие функции (*collect_hosts* и *display_hosts*) выполняются отсоединенными потоками немедленно. Для элементов 2 и 3 соответствующие функции (*getcmd* и *quit_prog*) отсоединенными потоками не выполняются. Каждая из четырех функций возвращает ненулевое значение, если она выполнена успешно, или 0, если она завершилась неудачно.

Потоки, созданные для выполнения функциями *collect_hosts* и *display _hosts*, отсоединяются от главных потоков. Как только они завершатся, их ресурсы и идентификаторы могут использоваться новыми потоками. Это объясняется тем, что интерактивная программа может работать довольно долго и должна на соответствующем уровне поддерживать скорость своей реакции на действия

пользователя. Она не может позволить себе приостановиться в вызове *thr_join* и ожидать завершения других потоков.

При вызове функций *collect_hosts* и *display_hosts* аргументы не указываются. Функция *getcmd* получает от пользователя команду shell и имя хост-машины. Затем она создает поток для выполнения функции *exec_shell*. Фактическое значение, передаваемое в функцию *exec_shell*,— символьная строка, завершающаяся символом NULL и содержащая команду, знак "/" и имя хост-машины, на которой должна быть выполнена команда. Например, если в функцию *exec_shell* передан аргумент

```
"cal 1995 > foo/fruit"
```

то она попросит выполнить команду *cal 1995 > foo* на машине *fruit*. Перечень хост-машин, на которых инсталлирован демон *shell_svc*, определяется функцией *collect_hosts* и сообщается пользователю функцией *display_hosts*.

Определения функций *collect_hosts*, *display_hosts* и *exec_shell* содержатся в файле *shell_cls.C*:

```
#include <iostream.h>
#include <fstream.h>
#include <strstream.h>
#include <stdio.h>
#include <netdir.h>
#include <rpc/rpc.h>
#include <string.h>
#include "mshell.h"
#include <thread.h>
#include "RPC.h"

#define MAXHOSTS   30
#define TMPFILE    "/tmp/hosts"

static char  *hostlist[MAXHOSTS];
static int   numhosts;
extern rwlock_t    rwlck; // определяется и инициализируется в main_shell.C

/* RPC-вызов сервера для выполнения одной команды */
int exec_host( const char* cmd, char* host )
{
    static int    res=0;
    RPC_cls cl( host, SHELLPROG, SHELLVER, "netpath");
    if (!cl.good()) return 1;

    cl.set_auth( AUTH_SYS );

    if (cl.call( EXECSHELL, (xdrproc_t)xdr_string, (caddr_t)&cmd,
            (xdrproc_t)xdr_int, (caddr_t)&res) != RPC_SUCCESS)
        return 2;

    if (res!=0) {
        cerr << "clnt: exec cmd fails\n";
        return 4;
    }
```



```
    return 0;
}


/* проверка наличия RPC-сервера на указанной хост-машине  */
int check_host( const char* hostnm )
{
    if (rw_rdlock(&rwlck)) perror("rw_rdlock");

    int i;
    for (i=0; i < numhosts; i++)
        if (!strcmp(hostlist[i],hostnm)) break;

    if (rw_unlock(&rwlck)) perror("rw_unlock");

    return  (i < numhosts) ? 1 : 0;
}

/* выполнить команду на удаленной хост-машине */
void* exec_shell( void* argp )
{
    static int rc = 0;
    char* cmd = (char*)argp;
    /* the input string syntax is: <cmd>'/'<host> */
    char* host = strrchr(cmd,'/');
    *host++ = '\0';
    if (!check_host(host)) {
        cout << "Invalid host: '" << host << "'\n" << flush;
        rc = 1;
        thr_exit(&rc);
    }
    rc = exec_host( cmd, host );
    thr_exit(&rc);
    return 0;
}

/* вывести список всех доступных хост-машин */
void* display_hosts( void* argp )
{
    int rc = 0;

    if (rw_tryrdlock(&rwlck)) {
        cout << "Host table in processed. please waite...\n";

        if (rw_rdlock(&rwlck)) perror("rw_rdlock");
    }

    if (!numhosts)  {
        cout << "Host table is empty!\n" << flush;
        cout << "please select 'Collect hosts info' option\n" << flush;
        rc = -1;
    }
    else {
        char buf[256];
        ostrstream(buf,256) << TMPFILE << "." << thr_self();
```

```
        ofstream ofs (buf);
        if (!ofs)
            cerr << "Create temp file '" << buf << "' failed\n";
        else {
            int i;
            for ( i=0; i < numhosts; i++)
                ofs << i << ":   " <<  hostlist[i] << endl;
            ofs.close();
            char cmd[256];
            ostrstream(cmd,256) << "xterm -title Hosts -e view " << buf;
            if (system(cmd)) perror("system");
            if (unlink(buf)) perror("unlink");
        }
    }

    if (rw_unlock(&rwlck)) perror("rw_unlock");
    thr_exit(&rc);
    return 0;
}

/* записать имя удаленной хост-машины в таблицу */
int add_host( const char* hostnm )
{
    int new_entry = 1;
    if (rw_wrlock(&rwlck)) perror("rw_wrlock");

    int i;
    for ( i=0; i numhosts; i++)
        if (!strcmp(hostlist[i],hostnm)) break;

    if (i >= numhosts) {
        if (numhosts >= MAXHOSTS)
            cerr << "Too many remote hosts detected\n";
        else {
            hostlist[numhosts] = new char[strlen(hostnm)+1];
            strcpy(hostlist[numhosts++],hostnm);
        }
    }
    else new_entry = 0;
    if (rw_unlock(&rwlck)) perror("rw_unlock");
    return new_entry;
}

/* клиентская широковещательная функция обратного вызова */
bool_t callme (caddr_t res_p, struct netbuf* addr, struct netconfig *nconf)
{
    int i;
    struct nd_hostservlist *servp;
    if (netdir_getbyaddr(nconf,&servp,addr))
        perror("netdir_getbyaddr");
    else for ( i=0; i < servp->h_cnt; i++)
        if (!add_host( servp->h_hostservs[i].h_host ))
            return TRUE;    /* закончить широковещательную рассылку,
                               если имя хост-машины встретилось
                               дважды */
```



```
    return FALSE;
}

/* определить удаленные хост-машины, на которых работает RPC-сервер */
void* collect_hosts( void* argp )
{
    /* клиент посылает широковещательный запрос и ждет ответа */
    int rc = RPC_cls::broadcast( SHELLPROG, SHELLVER, 0,
                                 (resultproc_t)callme,
                (xdrproc_t)xdr_void, (caddr_t)NULL,
                (xdrproc_t)xdr_void, (caddr_t)NULL);

    switch (rc) {
        case RPC_SUCCESS:
            break;
        case RPC_TIMEDOUT:
            if (numhosts) break;
        default:
            cerr << "RPC broadcast failed\n";
            rc = 1;
    }


    thr_exit(&rc);
    return 0;
}
```

Функция *collect_hosts* выполняет широковещательный RPC-вызов для опроса всех демонов *shell_svc* в сети. Каждый ответ демона на RPC-вызов регистрируется функцией *callme*. Эта функция извлекает имя сервера с помощью функции *netdir_getbyaddr* и добавляет это имя в таблицу *hostlist* с помощью функции *add_host*, которая возвращает 1 в случае успеха. В противном случае возвращается 0. Функция *callme* завершает широковещательный RPC, если видит, что одна и та же хост-машина ответила на запрос дважды. Это означает, что новых ответов нет и процесс широковещательной передачи повторяется.

По завершении широковещательной передачи RPC-функция *collect_hosts* проверяет результат и закрывает свой поток. В случае успеха она возвращает RPC_SUCCESS, а в случае неудачи — ненулевое значение.

Контроль доступа к таблице *hostlist* выполняется с помощью блокировки чтения-записи *rwlck*. Это необходимо потому, что к таблице *hostlist* обращаются все потоки, созданные для выполнения функций *collect_hosts*, *display_hosts* и *exec_shell*. Важно, чтобы потоки записи (выполняющие функцию *collect_hosts*) не обращались к таблице *hostlist* одновременно с потоками чтения (выполняющими функции *display_hosts* и *exec_shell*).

Активизированная функция *display_hosts* отображает все имена, имеющиеся в таблице *hostlist* (это имена хост-машин, на которых работает демон *shell_svc*). Функция *display_hosts* устанавливает блокировку чтения-записи перед обращением к таблице *hostlist* и снимает ее, завершив обращение, чтобы потоки *collect_hosts* не могли внести изменения в таблицу во время чтения.

Чтобы отделить вывод списка хост-машин от отображения меню главного потока, функция *display_hosts* сохраняет список имен во временном файле и вызывает программу *xterm* для выполнения команды *view* <*temp_file*> в отдельном окне. Это окно исчезает, когда пользователь завершает работу программы *view* и временный файл удаляется. В случае успешного выполнения функция закрывает поток с кодом возврата 0, а в случае неудачи — с ненулевым кодом.

Входным аргументом при вызове команды *exec_shell* является символьная строка, которая содержит определенную пользователем команду, имя хост-машины и завершается символом NULL. Функция вызывает команду *check _host*, которая проверяет, находится ли указанное имя в таблице. Если имени в таблице нет, поток закрывается с кодом 1 (неудача). Если имя в таблице есть, функция вызывает функцию *exec_host*, которая создает объект *RPC_cls* для установления соединения с демоном *shell_svc* на указанной хост-машине. Функция *exec_host* вызывает RPC-функцию *execshell*, которая выполняет пользовательскую команду на указанной машине. Функция *exec_host* и поток *exec_shell* завершаются с нулевым значением, если команда на удаленной машине выполнена нормально; в противном случае возвращается ненулевое значение.

Наша интерактивная программа состоит из файлов *main_shell.C* и *shell _cls.C*. Ниже показано, как она компилируется и запускается. В приведенном примере *fruit* и *veggie* — имена хост-машин, на которых инсталлирован демон *shell_svc*.

```
% CC -DSYSV4 shell_cls.C main_shell.C -o shell_cls -lthread -lnsl
% shell_cls
0: Collect hosts names
1: Display hosts names
2: Execute command
3: Quit
Selection> 0

0: Collect hosts names
1: Display hosts names
2: Execute command
3: Quit
Selection> 1
<в окне программы xterm выводится список: 0:fruit 1:veggie>

0: Collect hosts names
1: Display hosts names
2: Execute command
3: Quit
Selection> 2

shell cmd> cal 1995 > foo
host> fruit
```



```
0: Collect hosts names
1: Display hosts names
2: Execute command
3: Quit
Selection> 3
```

13.9. Заключение

В этой главе рассмотрены методы многопотокового программирования, основанные на библиотеке потоков ОС Solaris фирмы Sun Microsystems и стандарте POSIX.1c. И в библиотеке Sun, и в стандарте POSIX.1c есть набор многопотоковых библиотечных функций, которые позволяют пользователям создавать в своих программах потоки выполнения и управлять ими. Кроме того, в распоряжении пользователей имеются различные объекты синхронизации: взаимоисключающие блокировки, условные переменные и семафоры,— с помощью которых пользователи могут синхронизировать доступ потоков к общим данным в одном процессе.

Среди других средств системной поддержки многопотоковых программ — специальные API, которые позволяют модифицировать сигнальные маски отдельных потоков, и реентерабельные версии основных библиотечных функций. Ожидается, что все эти средства будут присутствовать и в других платформах, поддерживающих многопотоковые программы.

Многопотоковое программирование особенно полезно для объектно-ориентированных приложений, выполняемых на многопроцессорных машинах. В этом можно убедиться на примере распределенной многопотоковой интерактивной программы, которая приведена в последнем разделе. Взяв ее за основу, пользователи могут создавать собственные приложения, используя многопроцессорные и сетевые вычислительные ресурсы, которыми обладают машины, выполняющие их приложения.

Предметный указатель

А

Аутентификация 475
— AUTH_DES 478
— AUTH_NONE 476
— AUTH_SYS 476
— AUTH_UNIX 476

Б

Библиотечные функции RPC:
— rstat 432
— rusers 432
— rwall 432
— spray 432

Блоки-ловушки 62

Блокировка файлов 173

Блокировки:
— записи (исключающие) 174
— обязательные 174
— продвижение 176
— разделение 176
— рекомендуемые 174
— чтения (разделяемые) 174

Г

Гнезда 365

Д

Данные:
— закрытые 32
— защищенные 32
— открытые 32
— статические 35
— члены 33

З

Заголовки 20

И

Индексные дескрипторы 135

Интервальные таймеры 278

Интерфейс транспортного уровня (TLI) 389

Исключительные ситуации 62

К

Классы:
— devfile 199
— dirfile 197
— filebase 191
— fstream 191
— message 310
— pipefile 198
— regfile 194
— RPC 450
— sock 378
— symfile 201
— timer 287
— TLI 409
— абстрактные 47
— базовые (надклассы) 40
— виртуальные базовые 44
— дружественные 38
— наследование 40
— объявление 33
— потока ввода-вывода 71
— производные (подклассы) 40
— шаблоны 58

Компилятор rpcgen 433

Конечные точки транспортировки 389

М

Манипуляторы 78

Межпроцессное взаимодействие 296
— разделяемая память:
POSIX.1b 315, 330, 355
UNIX System V 297, 321, 333

Многопотоковое программирование 513

О

Обратный вызов 493

Объекты синхронизации 533
— взаимоисключающие блокировки 534
— семафоры 550
— условные переменные 540

Операторы:
— throw 62

Операции:
— new 50
— перегрузка 53
— расширения области видимости 34

П

Процессы 207

С

Сигналы 259

Список генерации 67

Список инициализации 43

Стандартные библиотечные функции С 85
— abort 94
— atexit 94
— atof 92
— atoi 92
— atol 92
— calloc 102
— crypt 120
— ctime 104
— encrypt 121
— endpwent 117
— exit 94
— fdopen 89
— fgetgrent 119
— fgetpwent 118
— free 102
— freopen 88
— getenv 94
— getgrent 119
— getgrgid 119
— getgrnam 119
— getgrwent 119
— getopt 113
— getpwnam 117
— getpwuid 117
— gmtime 105
— localtime 105
— malloc 101
— memccpy 101
— memchr 101
— memcmp 100
— memcpy 100
— memset 99
— pclose 90
— popen 90
— putenv 94
— rand 93
— realloc 102
— setjmp 115
— setkey 121
— setpwent 117
— srand 93
— strcat 95
— strchr 95
— strcmp 95
— strcpy 95
— strerror 98
— strlen 95
— strncat 95
— strncmp 95
— strncpy 95
— strnspn 96
— strpbrk 95
— strrchr 95
— strspn 96
— strstr 95
— strtod 92
— strtok 97
— strtol 92
— strtoul 92
— system 91
— time 104

Т

Точки транспортировки конечные 389

У

Удаленные вызовы процедур (RPC) 429

Ф

Файловые API 147
— FIFO-файлов:
mkfifo 185
pipe 188
— каталогов:
closedir 181
mknod 179
opendir 180
readdir 180
rewinddir 181
rmdir 181
seekdir 181
telldir 181
— общие 148
access 167
chmod 168
chown 170
close 155
creat 152
fchmod 168
fchown 170
fcntl 156, 175
fstat 161
lchown 170
link 159
lseek 158
open 149
read 153
stat 161
unlink 160
utime 172
write 154
— символических ссылок:
lstat 190
readlink 189
symlink 188, 189
mknod 183

Файлы заголовков 20

Файлы 127
— жесткие ссылки 143
— каталоги 142
— символические ссылки 143

Функции:
— виртуальные 43
— встроенные 36
— деструктор 33, 36
— дружественные 38
— конструктор 33, 36
— непостоянные 39
— постоянные 39
— статические 35
— чистые виртуальные 47
— члены 33
— шаблоны 56

А

API TLI:

— t_accept 399
— t_bind 396
— t_close 408
— t_connect 401
— t_listen 398
— t_open 393
— t_rcv 404
— t_rcvconnect 401
— t_rcvdis 407
— t_rcvrel 407
— t_rcvudata 405
— t_rcvuderr 405
— t_snd 403
— t_snddis 407
— t_sndrel 406
— t_sndudata 403

API ввода-вывода с отображением в памяти:

— mmap 348
— msync 351
— munmap 351

API гнезд:

— accept 370
— bind 368
— connect 369
— listen 369
— recv 372
— recvfrom 372
— send 371
— sendto 371
— shutdown 373
— socket 367

API потоков выполнения:

— pthread_create 529
— pthread_detach 531
— pthread_exit 531
— pthread_join 531
— pthread_kill 531
— pthread_sigmask 531
— sched_yield 532
— thr_continue 520
— thr_create 519
— thr_exit 521
— thr_getconcurrency 523
— thr_getprio 523
— thr_join 521
— thr_kill 522
— thr_min_stack 520
— thr_self 520
— thr_setconcurrency 523
— thr_setprio 523
— thr_sigsetmask 522
— thr_suspend 520
— thr_yield 523

API процессов:

— _exit 215
— fork 212
— getegid 240
— geteuid 240
— getpgrp 239
— getpid 239
— getppid 239
— pipe 225
— setegid 242
— seteuid 242
— setgid 242
— setpgid 241
— setsid 241
— setuid 242
— vfork 212
— wait 216
— waitpid 216

API разделяемой памяти:

— ftruncate 355
— semctl 339
— shm_open 355
— shm_unlink 356
— shmat 337
— shmdt 338
— shmget 336

API семафоров:

— sem_close 332
— sem_destroy 332
— sem_getvalue 331
— sem_init 331
— sem_open 331
— sem_post 332
— sem_trywait 332
— sem_unlink 332
— sem_wait 332
— semctl 328
— semget 326
— semop 327

API сигналов:

— alarm 276
— kill 274
— sigaction 268
— sigfillset 266
— sighold 268
— sigismember 266
— signal 263
— sigpause 268
— sigpending 267
— sigprocmask 265
— sigset 264

API сообщений:

— mq_close 317
— mq_getattr 318
— mq_notify 317
— mq_receive 316
— mq_send 316
— mq_setattr 318
— msgctl 307
— msgget 303
— msgrcv 305
— msgsnd 304

API удаленных вызовов процедур:

— clnt_call 470
— clnt_create 469
— clnt_pcreateerror 470
— clnt_perror 470
— svc_create 467
— svc_getargs 468
— svc_run 468
— svc_sendreply 469
— svctcp_create 468
— svcudp_create 468

Содержание

Предисловие 5

Глава 1. Операционная система UNIX и стандарты ANSI 11

1.1. Стандарт ANSI C 12
1.2. Стандарт ANSI/ISO C++ 17
1.3. Различия между ANSI C и C++ 17
1.4. Стандарты POSIX 18
1.4.1. Среда POSIX 20
1.4.2. Макрокоманды тестирования характеристик по стандарту POSIX 21
1.4.3. Проверка ограничений во время компиляции и во время выполнения 23
1.5. Стандарт POSIX.1 FIPS 28
1.6. Стандарты X/Open 29
1.7. Заключение 29
1.8. Литература 30

Глава 2. Обзор языка C++ 31

2.1. Средства объектно-ориентированного программирования в C++ 32
2.2. Объявление классов в C++ 33
2.3. Дружественные функции и классы 38
2.4. Функции-члены, объявленные с декларацией const 39
2.5. Наследование классов в C++ 40
2.6. Виртуальные функции 43
2.7. Виртуальные базовые классы 44
2.8. Абстрактные классы 47
2.9. Операции new и delete 50
2.10. Перегрузка операций 53
2.11. Шаблоны функций и шаблоны классов 55
2.11.1. Шаблоны функций 56
2.11.2. Шаблоны классов 58
2.12. Обработка исключительных ситуаций 62
2.12.1. Исключительные ситуации и соответствие им блоков-ловушек 66
2.12.2. Объявление функций с оператором throw 67
2.12.3. Функции terminate и unexpected 67
2.13. Заключение 68
2.14. Литература 69

Глава 3. Классы потоков ввода-вывода языка C++ 71

3.1. Классы ввода-вывода в стандартные потоки 72
3.1.1. Класс istream 72
3.1.2. Класс ostream 74
3.1.3. Класс iostream 75
3.1.4. Класс ios 75
3.2. Манипуляторы 78
3.3. Классы ввода-вывода файлов 79

3.4. Классы strstream 81
3.5. Заключение 83

Глава 4. Стандартные библиотечные функции C 85

4.1. <stdio.h> 87
4.2. <stdlib.h> 91
4.3. <string.h> 95
4.3.1. strspn, strcspn 96
4.3.2. strtok 97
4.3.3. strerror 98
4.4. <memory.h> 99
4.5. <malloc.h> 101
4.6. <time.h> 104
4.7. <assert.h> 106
4.8. <stdarg.h> 107
4.9. Аргументы командной строки и ключи 112
4.10. <setjmp.h> 114
4.11. <pwd.h> 116
4.12. <grp.h> 118
4.13. <crypt.h> 120
4.14. Заключение 122

Глава 5. Интерфейсы прикладного программирования UNIX и POSIX 123

5.1. Интерфейсы прикладного программирования POSIX 124
5.2. Среда разработки UNIX и POSIX 124
5.3. Общие характеристики интерфейсов прикладного программирования 125
5.4. Заключение 26

Глава 6. Файловая система ОС UNIX 127

6.1. Типы файлов 128
6.2. Файловые системы UNIX и POSIX 131
6.3. Атрибуты файлов в UNIX и POSIX 132
6.4. Индексные дескрипторы в UNIX System V 134
6.5. Интерфейсы прикладного программирования для файлов 136
6.6. Поддержка файлов ядром UNIX 137
6.7. Взаимосвязь указателей потоков C и дескрипторов файлов 140
6.8. Каталоги 142
6.9. Жесткие и символические ссылки 143
6.10. Заключение 146

Глава 7. Файловые API операционной системы UNIX 147

7.1. Общие файловые API 148
7.1.1. Функция open 148
7.1.2. Системный вызов creat 152
7.1.3. Функция read 152
7.1.4. Функция write 154
7.1.5. Функция close 155
7.1.6. Функция fcntl 155
7.1.7. Функция lseek 157
7.1.8. Функция link 159
7.1.9. Функция unlink 160
7.1.10. Функции stat и fstat 161
7.1.11. Функция access 167
7.1.12. Функции chmod, fchmod 168
7.1.13. Функции chown, fchown, lchown 170
7.1.14. Функция utime 172
7.2. Блокировка файлов и записей 173
7.3. API каталогов 178
7.4. API файлов устройств 182

7.5. API FIFO-файлов 185
7.6. API символических ссылок 188
7.7. Общий класс для файлов 191
7.8. Класс regfile для обычных файлов 194
7.9. Класс dirfile для каталогов 197
7.10. Класс pipefile для FIFO-файлов 198
7.11. Класс devfile для файлов устройств 199
7.12. Класс symfile для символических ссылок 201
7.13. Программа вывода на экран списка файлов 202
7.14. Заключение 205

Глава 8. Процессы операционной системы UNIX **207**

8.1. Поддержка процессов ядром ОС UNIX 208
8.2. API процессов 212
8.2.1. Функции fork, vfork 212
8.2.2. Функция _exit 215
8.2.3. Функции wait, waitpid 216
8.2.4. Функция exec 220
8.2.5. Функция pipe 225
8.2.6. Переназначение ввода-вывода 228
8.3. Атрибуты процессов 238
8.4. Изменение атрибутов процесса 241
8.5. Программа minishell 242
8.6. Заключение 257

Глава 9. Сигналы **259**

9.1. Поддержка сигналов ядром ОС UNIX 261
9.2. Функция signal 263
9.3. Сигнальная маска 265

9.4. Функция sigaction 268
9.5. Сигнал SIGCHLD и API waitpid 271
9.6. API sigsetjmp и siglongjmp 272
9.7. API kill 274
9.8. API alarm 276
9.9. Интервальные таймеры 277
9.10. Таймеры стандарта POSIX.1b 281
9.11. Класс timer 287
9.12. Заключение 293

Глава 10. Межпроцессное взаимодействие **295**

10.1. Методы IPC, соответствующие стандарту POSIX.1b 296
10.2. Методы IPC, применяющиеся в UNIX System V 297
10.3. Сообщения в UNIX System V 297
10.3.1. Поддержка сообщений ядром UNIX 298
10.3.2. API ОС UNIX, предназначенные для обмена сообщениями 300
10.3.3. Функция msgget 303
10.3.4. Функция msgsnd 304
10.3.5. Функция msgrcv 305
10.3.6. Функция msgctl 307
10.3.7. Пример приложения клиент/сервер 308
10.4. Сообщения в стандарте POSIX.1b 315
10.4.1. Класс message стандарта POSIX.1b 318
10.5. Семафоры в UNIX System V 321
10.5.1. Поддержка семафоров ядром UNIX 322
10.5.2. API ОС UNIX для семафоров 324
10.5.3. Функция semget 326
10.5.4. Функция semop 327
10.5.5. Функция semctl 328
10.6. Семафоры POSIX.1b 330
10.7. Разделяемая память в UNIX System V 333

10.7.1. Поддержка разделяемой памяти ядром UNIX 333
10.7.2. API ОС UNIX для разделяемой памяти 335
10.7.3. Функция shmget 336
10.7.4. Функция shmat 337
10.7.5. Функция shmdt 338
10.7.6. Функция semctl 339
10.7.7. Пример приложения клиент/сервер с семафорами и разделяемой памятью 341
10.8. Ввод-вывод с отображением в память 347
10.8.1. API ввода-вывода с отображением в память 348
10.8.2. Функция mmap 348
10.8.3. Функция munmap 351
10.8.4. Функция msync 351
10.8.5. Программа типа клиент/сервер, использующая функцию mmap 352
10.9. Организация разделяемой памяти в соответствии со стандартом POSIX.1b 355
10.9.1. Программа типа клиент/сервер, соответствующая стандарту POSIX.1b 357
10.10. Заключение 362

Глава 11. Гнезда и интерфейс транспортного уровня **363**

11.1. Гнезда 364
11.1.1. Функция socket 367
11.1.2. Функция bind 368
11.1.3. Функция listen 369
11.1.4. Функция connect 369
11.1.5. Функция accept 370
11.1.6. Функция send 370
11.1.7. Функция sendto 371
11.1.8. Функция recv 372
11.1.9. Функция recvfrom 372
11.1.10. Функция shutdown 373
11.2. Создание потоковых гнезд 373

11.3. Пример приложения типа клиент/сервер, предназначенного для обработки сообщений 385
11.4. Интерфейс транспортного уровня (TLI) 389
11.4.1. API интерфейса транспортного уровня 390
11.4.2. Функция t_open 393
11.4.3. Функция t_bind 396
11.4.4. Функция t_listen 398
11.4.5. Функция t_accept 399
11.4.6. Функция t_connect 401
11.4.7. Функции t_snd, t_sndudata 402
11.4.8. Функции t_rcv, t_rcvudata и t_rcvuderr 404
11.4.9. Функции t_sndrel, t_rcvrel 406
11.4.10. Функции t_snddis, t_rcvdis 407
11.4.11. Функция t_close 408
11.5. Класс TLI 409
11.6. Пример передачи сообщений по схеме клиент/сервер 416
11.7. Пример взаимодействия процессов с помощью дейтаграмм 421
11.8. Заключение 426

Глава 12. Удаленные вызовы процедур **429**

12.1. История создания RPC 430
12.1.1. Уровни интерфейса программирования RPC 430
12.2. Библиотечные функции RPC 431
12.3. Компилятор rpcgen 433
12.3.1. Функция clnt_create 438
12.3.2. Программа rpcgen 439
12.3.3. Получение списка файлов каталога с помощью программы rpcgen 440
12.3.4. Недостатки компилятора rpcgen 444
12.4. Низкоуровневый интерфейс программирования RPC 445
12.4.1. Функции XDR-преобразования 445
12.4.2. Низкоуровневые API удаленных вызовов процедур 447
12.5. Классы RPC 449

12.5.1. Функция svc_create . . . 467
12.5.2. Функция svc_run . . . 468
12.5.3. Функция svc_getargs . . . 468
12.5.4. Функция svc_sendreply . . . 469
12.5.5. Функция clnt_create . . . 469
12.5.6. Функция clnt_call . . . 470

12.6. Управление набором RPC-программ и версий . . . 471

12.7. Аутентификация . . . 475
12.7.1. Метод AUTH_NONE . . . 476
12.7.2. Метод AUTH_SYS (или AUTH_UNIX) . . . 476
12.7.3. Метод AUTH_DES . . . 478
12.7.4. Получение списка файлов каталога с применением аутентификации . . . 481

12.8. Широковещательный режим RPC . . . 489
12.8.1. Пример широковещательной передачи RPC-запросов . . . 491

12.9. Обратный вызов RPC . . . 493

12.10. Временный номер RPC-программы . . . 499

12.11. RPC-услуги на базе inetd . . . 504

12.12. Заключение . . . 511

Глава 13. Многопотоковое программирование 513

13.1. Структура и методика использования потоков выполнения . . . 515

13.2. Потоки и облегченные процессы . . . 516

13.3. API потоков выполнения фирмы Sun Microsystems . . . 518
13.3.1. Функция thr_create . . . 519
13.3.2. Функции thr_suspend, thr_continue . . . 520
13.3.3. Функции thr_exit, thr_join . . . 521
13.3.4. Функции thr_sigsetmask и thr_kill . . . 521
13.3.5. Функции thr_setprio, thr_getprio и thr_yield . . . 523
13.3.6. Функции thr_setconcurrency и thr_getconcurrency . . . 523
13.3.7. Пример многопотоковой программы . . . 524

13.4. API потоков выполнения, определенный в стандарте POSIX.1c . . . 528

13.4.1. Функция pthread_create . . . 529
13.4.2. Функции pthread_exit, pthread_detach и pthread_join . . . 531
13.4.3. Функции pthread_sigmask и pthread_kill . . . 531
13.4.4. Функция sched_yield . . . 532

13.5. Объекты синхронизации потоков выполнения . . . 533
13.5.1. Взаимоисключающие блокировки . . . 534
13.5.1.1. Взаимоисключающие блокировки Sun . . . 534
13.5.1.2. Взаимоисключающие блокировки POSIX.1c . . . 535
13.5.1.3. Примеры взаимоисключающих блокировок . . . 536
13.5.2. Условные переменные . . . 540
13.5.2.1. Условные переменные фирмы Sun . . . 541
13.5.2.2. Пример программы с условными переменными . . . 542
13.5.2.3. Условные переменные стандарта POSIX.1c . . . 545
13.5.3. Блокировки чтения-записи фирмы Sun . . . 546
13.5.4. Семафоры . . . 550

13.6. Данные потоков . . . 554

13.7. Среда многопотокового программирования . . . 561

13.8. Пример распределенного многопотокового приложения . . . 562

13.9. Заключение . . . 573

Предметный указатель 575

Книги издательства

Для специалистов:

Э. Немет. **UNIX: руководство системного администратора.** 2-е изд. 832 с.

С. Спейнаур, В. Куэрсиа. **Справочник Web-мастера.** 350 с.

К. Хант. **Персональные компьютеры в сетях TCP/IP.** 384 с.

Г. Шилдт. **MFC: основы программирования.** 560 с.

Й. Шиб, М. Мюллер. **Windows NT 4 Workstation.** 768 с.

Р. Петерсен. **LINUX: руководство по операционной системе** (в комплекте с CD-ROM). 2-е изд. 688 с.

Д. Мейнджер. **JAVA: основы программирования** (в комплекте с CD-ROM). 320 с.

С. Пек, С. Аррантс. **Web-сервер WebSite** (в комплекте с CD-ROM). 544 с

Д. Рули и др. **Сети Windows NT 4.0.** 800 с.

Д. Мейнджер. **JavaScript: основы программирования** (в комплекте с дискетой). 320 с.

Б. Мейсо. **Visual J++: основы программирования.** 400 с.

Д. Мюррей, В. Риппер. **Энциклопедия форматов графических файлов** (в комплекте с CD-ROM). 672 с.

Б. Лори, П. Лори. **Apache** (в комплекте с CD-ROM). 288 с.

Э. Титтел и др. **Создание VRML-миров** (в комплекте с CD-ROM). 320 с.

Для пользователей:

В. Пасько. **Word 97.** 2-е изд. 480 с.

А. Колесников. **Excel 97.** 2-е изд. 528 с.

В. Пасько. **Access 97.** 416 с.

BHV Publishing Group

В ближайшее время увидят свет:

Д. Пьюполо. **OLE: создание элементов управления.** 432 с.

В. Пасько и др. **Office 97** (русская версия). 600 с.

С. Томас. **Создание intranet-сети средствами Windows NT 4.0.** 400 с.

А. Колесников. **Excel 97** (русская версия). 512 с.

В. Пасько. **Word 97** (русская версия). 432 с.

Р. Шварц, Т. Кристиансен. **Изучаем Perl.** 400 с.

В. Пасько. **Access 97** (русская версия). 432 с.

По вопросам приобретения книг обращайтесь к нашим торговым представителям:

- **г. Москва: "СПАРРК"**
  тел./факс (095) 148-35-12, e-mail: serg@sparrk.msk.ru
- **г. Москва: "АСКОЛЬД"**
  тел. (095) 148-94-80, магазин "Книги"
- **г. Киев: Издательство "Ирина"**
  тел./факс (044) 269-04-23, e-mail: market@bhv.kiev.ua

За более подробной информацией обращайтесь на Web-сервер нашего издательства *www.bhv.kiev.ua*

Учебное пособие

Теренс Чан

Системное программирование на C++ для UNIX

Редакторы В.С. Гусев, И.В. Карпышенко
Технический редактор О.Н. Заплаткина

Лицензия на издательскую деятельиость № 071405
от 28 февраля 1997 г.

ООО «Спаррк».
123364, Москва, ул. Свободы, д. 28, корп. 2.

Подписаио в печать 09.10.97. Формат 70×100 1/16.
Печать офсетная. Усл. печ. л. 37. Тираж 5000 экз.
Заказ № 933.

Отпечатано с диапозитивов в ГПП «Печатиый Двор»
Государствеииого комитета РФ по печати.
197110, Саикт-Петербург, Чкаловский пр., 15.