Advanced Programming in the UNIX® Environment

Second Edition

W. Richard Stevens,
Stephen A. Rago

Addison-Wesley

HIGH TECH

UNIX

Профессиональное программирование

Второе издание

У. Ричард Стивенс,
Стивен А. Раго

Санкт-Петербург — Москва
2007

Серия «High tech»

У. Ричард Стивенс, Стивен А. Раго

UNIX. Профессиональное программирование, 2-е издание

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*М. Деркачев*
Редактор	*Р. Павлов*
Корректор	*Н. Ткачева*
Верстка	*Д. Орлова*

Стивенс Р., Раго С.
UNIX. Профессиональное программирование, 2-е издание. – СПб.: Символ-Плюс, 2007. – 1040 с., ил.
ISBN 5-93286-089-8

«UNIX. Профессиональное программирование» представляет собой подробнейшее справочное руководство для любого профессионального программиста, работающего с UNIX. Стивену Раго удалось обновить и дополнить текст фундаментального классического труда Стивенса, сохранив при этом точность и стиль оригинала. Содержание всех тем, примеров и прикладных программ обновлено в соответствии с последними версиями наиболее популярных реализаций UNIX. Среди важных дополнений главы, посвященные потокам и разработке многопоточных программ, использованию интерфейса сокетов для организации межпроцессного взаимодействия (IPC), а также широкий охват интерфейсов, добавленных в последней версии POSIX.1. Аспекты прикладного программного интерфейса разъясняются на простых и понятных примерах, протестированных на 4-х платформах: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9 и Mac OS X 10.3. Описывается множество ловушек, о которых следует помнить при написании программ для различных реализаций UNIX, и показывается, как их избежать, опираясь на стандарты POSIX.1 и Single UNIX Specification.

ISBN-13: 978-5-93286-089-2
ISBN-10: 5-93286-089-8
ISBN 0-201-43307-9 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 18.04.2007. Формат 70x100 1/16 . Печать офсетная.
Объем 65 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Посвящается Джин

Оглавление

Отзывы ко второму изданию

«Обновление, выполненное Стивеном Раго (Stephen Rago), – это событие, которого давно и с нетерпением ждало все сообщество профессионалов, использующих в своей работе многоликое семейство UNIX и UNIX-подобных операционных систем. В этом издании исключены устаревшие и добавлены новейшие сведения. Содержание всех тем, примеров и прикладных программ обновлено в соответствии с последними версиями наиболее популярных реализаций UNIX и UNIX-подобных операционных систем. И кроме того, при этом полностью сохранен стиль изложения оригинала».

– Мукеш Кэкер (Mukesh Kacker),
соучредитель и бывший технический директор Pronto Networks, Inc.

«Один из фундаментальных классических трудов, посвященных программированию для UNIX».

– Эрик С. Рэймонд (Eric S. Raymond),
автор книги «The Art of UNIX Programming»

«Это издание представляет собой подробнейшее справочное руководство для любого профессионального программиста, работающего с UNIX. Стивену Раго удалось обновить и дополнить текст классического произведения Стивенса, сохранив при этом точность оригинала. Аспекты прикладного программного интерфейса разъясняются на простых и понятных примерах. В книге также описывается множество ловушек, о которых следует помнить при написании программ для различных реализаций UNIX, и показывается, как их избежать, опираясь на соответствующие стандарты, такие как POSIX 1003.1 (редакция от 2004 года) и Single UNIX Specification, Version 3».

– Эндрю Джози (Andrew Josey), директор по сертификации
The Open Group и председатель рабочей группы POSIX 1003.1

«Второе издание книги – жизненно необходимый справочник для любого, кто занимается разработкой программ для UNIX. Эту книгу я открываю первой, когда хочу изучить или вспомнить какие-либо из интерфейсов системы. Стивен Раго удачно переработал содержание книги и включил в нее сведения о новейших операционных системах, таких как GNU/Linux и Apple OS X, придерживаясь при этом стиля первого издания – как в смысле удобочитаемости, так и в смысле полноты изложения. Для нее всегда найдется место рядом с моим компьютером».

– Доктор Бенджамин Куперман (Dr. Benjamin Kuperman),
колледж г. Свортмора (Swarthmore)

Отзывы к первому изданию

«Книга «Advanced Programming in the UNIX® Environment» обязательно должна быть у любого серьезного программиста, который пишет для UNIX на языке C. По своей основательности, глубине и ясности подачи материала она не имеет себе равных».

– UniForum Monthly

«Многочисленные читатели рекомендовали мне книгу «Advanced Programming in the UNIX® Environment», написанную Ричардом Стивенсом (издательство Addison-Wesley), и я благодарен им за это. Раньше я даже не слышал об этой книге, хотя она вышла в свет в 1992 году. Получив экземпляр книги, я с первых же глав был очарован ею».

– Open Systems Today

«Очень понятное и подробное описание внутреннего устройства UNIX вы найдете в книге «Advanced Programming in the UNIX® Environment», написанной Ричардом Стивенсом (Addison-Wesley). Она включает в себя множество практических примеров, и я нахожу ее очень полезной при разработке системного программного обеспечения».

– RS/Magazine

Вступительное слово

Почти в каждом интервью или после лекций в какой-то момент мне задают один и тот же вопрос: «Ожидали ли вы, что UNIX продержится так долго?». Разумеется, в ответ я говорю одно и то же: «Нет, для нас это оказалось полной неожиданностью». Некоторые даже подсчитали, что система в том или ином виде существует уже более половины всей жизни компьютерной индустрии.

Процесс развития был бурным и сложным. С начала 70-х годов прошлого столетия компьютерные технологии сильно изменились, особенно за счет глобальных сетевых технологий, вездесущей графики и широкого распространения персональных компьютеров, тем не менее система сумела учесть и вобрать в себя все эти явления. Несмотря на то, что сегодня в области настольных систем доминируют Microsoft и Intel, рынок в определенной степени двигается в направлении от единого поставщика к нескольким, а в последние годы все более ориентируется на открытые стандарты и свободно распространяемые системы.

К счастью, система UNIX, которую следует рассматривать как явление, а не только как торговую марку, не просто двигалась вперед, но и сумела занять лидирующее положение. В 70-х и 80-х годах XX века корпорация AT&T была держателем авторских прав на исходные тексты UNIX, но она всячески поощряла усилия по стандартизации, основанные на системных интерфейсах и языках. Например, AT&T опубликовала SVID (System V Interface Definition, описание интерфейса System V), которое легло в основу стандарта POSIX и последующих его модификаций. Так случилось, что UNIX смогла достаточно изящным образом приспособиться к работе в сетевом окружении и, может быть, менее элегантно, но все-таки на достаточно приемлемом уровне к работе с графикой. Кроме того, основные интерфейсы ядра UNIX и инструментальные средства уровня пользователя стали технологической основой движения за программное обеспечение, распространяемое с открытым исходным кодом.

Существенно то, что статьи и книги, посвященные системе UNIX, всегда были востребованы, даже в то время, когда программное обеспечение самой системы было запатентовано. Примером может служить книга Мориса Баха (Maurice Bach) «The Design of the Unix Operating System». Честно говоря, я мог бы утверждать, что основная причина такой долговечности системы состоит в ее привлекательности для талантливых авторов, которые стремились объяснить ее красоты и тайны. Брайан Керниган (Brian Kernighan) – один из них, а другой – конечно же, Рич Стивенс (Rich Stevens). Первое издание этой книги, а также серия его книг, посвященных сетевым технологи-

ям, справедливо считаются одними из лучших работ и потому пользуются заслуженной популярностью.

Первое издание этой книги вышло в свет еще до того, как получили широкое распространение Linux и другие реализации UNIX с открытыми исходными текстами, берущие свое начало из Беркли, а также когда большинство людей имели лишь модемное подключение к сети. Стив Раго (Steve Rago) тщательно выполнил обновление этой книги, чтобы учесть изменения, произошедшие в компьютерных технологиях и в стандартах ISO и IEEE с момента выхода первой публикации. Поэтому все примеры в книге обновлены и вновь протестированы.

Это самое достойное второе издание классики.

Деннис Ритчи (Dennis Ritchie)
Мюррей Хилл, Нью Джерси
Март 2005

Предисловие ко второму изданию

Введение

Мой первый контакт с Ричем Стивенсом состоялся по электронной почте, когда я сообщил ему об опечатке в его книге «UNIX Network Programming». Позже он говорил, что я оказался первым, кто прислал ему сообщение о найденной ошибке. До самой его смерти в 1999 году мы время от времени обменивались электронными письмами. Обычно это происходило, когда один из нас задавался каким-либо вопросом и полагал, что другой мог бы на него ответить. Мы встречались с ним за обедом на конференциях USENIX и на лекциях, которые он читал.

Рич Стивенс был другом и настоящим джентльменом. Когда в 1993 году я написал книгу «UNIX System V Network Programming», я подразумевал, что она является версией книги Рича «UNIX Network Programming», ориентированной на System V. Благодаря своему характеру Рич охотно взялся за рецензирование моих глав, воспринимая меня не как конкурента, но как коллегу. Мы часто говорили о сотрудничестве над версией его книги «TCP/IP Illustrated», посвященной STREAMS. Если бы события сложились по-иному, вероятно, мы сделали бы это, но Ричарда больше нет с нами, поэтому обновление данной книги я рассматриваю как самую тесную совместную нашу работу.

Когда издательство Addison-Wesley сообщило мне, что оно заинтересовано в обновлении книги Рича, я думал, что дополнений будет не очень много. Даже по прошествии 13 лет его работа остается достаточно актуальной. Но современный мир UNIX значительно отличается от того, каким он был во время выхода первого издания книги.

- Версии System V постепенно вытесняются операционной системой Linux. Основные производители аппаратного обеспечения, поставляющие свою продукцию в комплекте с собственными версиями UNIX, либо выпустили версии своих продуктов для Linux, либо объявили о ее поддержке. ОС Solaris, вероятно, осталась последней наследницей System V Release 4, обладающей более или менее заметной долей рынка.
- После выхода 4.4BSD группа по проведению исследований в области информационных технологий (CSRG – Computing Science Research Group) из Калифорнийского университета в Беркли приняла решение о завершении разработки операционной системы UNIX, однако существует несколько групп добровольцев, которые продолжают осуществлять поддержку общедоступных версий.

- Появление ОС Linux, поддерживаемой тысячами добровольцев, позволило любому обладать компьютером, оборудованным самыми новейшими устройствами и работающим под управлением UNIX-подобной операционной системы, распространяемой с исходными текстами. Успех Linux выглядит не совсем обычно, учитывая, что существует несколько свободно распространяемых BSD-альтернатив.
- В очередной раз проявив себя в качестве инновационной компании, Apple Computer отказалась от устаревшей операционной системы Mac и заменила ее системой, созданной на основе Mach и FreeBSD.

В связи с этим я попытался дополнить сведения, содержащиеся в книге, чтобы охватить эти четыре платформы.

Когда в 1992 году Рич написал свою книгу «Advanced Programming in the UNIX Environment», я избавился от почти всех руководств по программированию в UNIX. С тех пор я держу на своем столе две книги: словарь и «Advanced Programming in the UNIX Environment». Надеюсь, что вы найдете это издание книги не менее полезным.

Изменения во втором издании

Работа Рича сохранила свою актуальность. Я старался не изменять оригинальное изложение материала, но слишком много всего произошло за последние 13 лет. Особенно это относится к стандартам, которые затрагивают программные интерфейсы UNIX.

Везде, где это было необходимо, я дополнил описания интерфейсов, которые изменились в результате деятельности по стандартизации. Особенно это заметно в главе 2, посвященной стандартам. В этом издании мы будем основываться на стандарте POSIX.1 от 2001 года как более универсальном по сравнению с версией 1990 года, на которой была основана первая редакция книги. Стандарт ISO C от 1990 года был изменен и дополнен в 1999 году, и некоторые изменения коснулись интерфейсов, описываемых стандартом POSIX.1.

Сегодня спецификация POSIX.1 охватывает намного большее количество интерфейсов. Основные спецификации Single UNIX Specification (опубликованные The Open Group, ранее X/Open) вошли в состав POSIX.1. Теперь POSIX.1 включает в себя ряд стандартов 1003.1 и некоторые из предварительных стандартов, опубликованных ранее.

В соответствии с этим я дополнил книгу новыми главами, охватывающими новые темы. Понятия потоков и многопоточных приложений очень важны, поскольку они предоставляют программистам более элегантный способ организации параллельных вычислений и асинхронной обработки.

Интерфейс сокетов теперь стал частью стандарта POSIX.1. Он обеспечивает единый интерфейс межпроцессного взаимодействия (IPC – Interprocess Communication), не зависящий от местонахождения процессов, и его обсуждение является естественным продолжением глав, посвященных IPC.

Я опустил рассмотрение большинства интерфейсов реального времени, которые появились в POSIX.1. Их лучше всего изучать по книгам, специально посвященным созданию приложений реального времени. Одну из таких книг вы найдете в библиографии.

Я изменил некоторые примеры в последних главах, чтобы приблизить их к задачам реальной жизни. Например, в наши дни не многие системы работают с PostScript-принтерами через последовательный или параллельный порт. Чаще встречается случай, когда доступ к таким принтерам осуществляется посредством сети, поэтому я изменил учебный пример взаимодействия с принтером так, чтобы учесть это обстоятельство.

Глава, посвященная взаимодействию с модемом, потеряла свою актуальность. Однако, чтобы оригинальный материал не был утерян окончательно, он выложен на веб-сайте книги в двух форматах: PostScript (*http://www.apuebook.com/lostchapter/modem.ps*) и PDF (*http://www.apuebook.com/lostchapter/modem.pdf*).

Все исходные тексты примеров из книги также доступны на сайте *www.apuebook.com*. Большая часть примеров была протестирована на четырех платформах:

1. FreeBSD 5.2.1 – системе, происходящей от 4.4BSD, работающей на процессоре Intel Pentium.
2. Linux 2.4.22 (дистрибутив Mandrake 9.2) – свободно распространяемой UNIX-подобной операционной системе, работающей на процессоре Intel Pentium.
3. Solaris 9 – происходящей от System V Release 4 системе от Sun Microsystems, работающей на 64-битном процессоре UltraSPARCIIi.
4. Darwin 7.4.0 – системе, основанной на FreeBSD и Mach, которая поддерживается Apple Mac OS X, версия 10.3, на процессоре PowerPC.

Благодарности

Первое издание этой книги, которое сразу же стало классикой, было полностью написано Ричем Стивенсом.

Вероятно, мне не удалось бы справиться с обновлением книги без поддержки моей семьи. Они стоически терпели груды разбросанных повсюду бумаг (определенно их было больше, чем обычно), монополизацию мною большинства компьютеров в доме и огромное количество времени, когда мое лицо было спрятано за терминалом. Моя супруга Джин (Jeanne) даже помогла мне, установив Linux на одну из тестовых машин.

Технические рецензенты предложили немало улучшений и исправлений и помогли удостовериться в правильности содержимого книги. Большое спасибо Дэвиду Бозуму (David Bausum), Дэвиду Борхему (David Boreham), Кейту Бостику (Keith Bostic), Марку Эллису (Mark Ellis), Филу Говарду (Phil Howard), Эндрю Джози (Andrew Josey), Мукешу Кэкеру (Mukesh Kacker), Брай-

ану Кернигану (Brian Kernighan), Бенгту Клебергу (Bengt Kleberg), Бену Куперману (Ben Kuperman), Эрику Рэймонду (Eric Raimond) и Энди Рудофу (Andy Rudoff).

Кроме того, я хотел бы поблагодарить Энди Рудофа за ответы на вопросы, касающиеся ОС Solaris, и Денниса Ритчи за то, что он, «зарывшись» в старые бумаги, отвечал на вопросы по истории UNIX. Еще раз хочу поблагодарить сотрудников издательства Addison-Wesley, с которыми было приятно работать. Спасибо Тиррелу Альбо (Tyrrell Albaugh), Мэри Франц (Mary Franz), Джону Фуллеру (John Fuller), Карен Геттман (Karen Gettman), Джессике Голдстейн (Jessica Goldstein), Норин Реджине (Noreen Regina) и Джону Уэйту (John Wait). Мои благодарности Эвелин Пайл (Evelyn Pyle) за отличную работу по техническому редактированию.

Я также буду благодарен всем читателям, кто пришлет по электронной почте свои комментарии, предложения и замечания об ошибках.

Уоррен, Нью Джерси
Апрель 2005

Стивен Раго
sar@apuebook.com

Предисловие к первому изданию

Введение

В этой книге описаны программные интерфейсы системы UNIX: интерфейс системных вызовов и многочисленные функции, предоставляемые стандартной библиотекой языка C. Она предназначена для всех, кто пишет программы, работающие под управлением UNIX.

Подобно большинству операционных систем, UNIX предоставляет работающим в ней программам разнообразные службы: открытие и чтение файлов, запуск новых программ, выделение областей памяти, получение текущего времени и т. д. Все это называется *интерфейсом системных вызовов* (*system call interface*). Дополнительно стандартная библиотека языка C предоставляет огромное количество функций, которые используются практически в любой программе, написанной на C (форматированный вывод значений переменных, сравнение строк и тому подобное).

Интерфейс системных вызовов и библиотечные функции традиционно описываются во втором и третьем разделах «Unix Programmer's Manual» (Руководства программиста UNIX). Эта книга не дублирует указанные разделы. В ней вы найдете примеры и пояснения, которые отсутствуют в упомянутом руководстве.

Стандарты UNIX

Быстрый рост количества версий UNIX, наблюдавшийся в 80-е годы, был урегулирован различными международными стандартами, которые стали появляться с конца 80-х. К ним относятся стандарт ANSI языка программирования C, семейство стандартов IEEE POSIX (которые продолжают развиваться и по сей день) и руководство по обеспечению переносимости X/Open.

Данная книга описывает эти стандарты. И не просто описывает, а рассматривает их применительно к популярным реализациям – System V Release 4 и грядущей 4.4BSD. Здесь представлены соответствующие действительности описания, которых зачастую недостает самим стандартам и книгам, которые только описывают стандарты.

Организация книги

Эта книга делится на шесть частей:

1. Обзор и введение в базовые концепции, связанные с программированием в UNIX, и в терминологию (глава 1). Обсуждение достижений в области стандартизации UNIX и различных реализаций UNIX (глава 2).
2. Ввод-вывод: небуферизованный ввод-вывод (глава 3), характеристики файлов и каталогов (глава 4), стандартная библиотека ввода-вывода (глава 5) и стандартные системные файлы (глава 6).
3. Процессы: окружение процессов в UNIX (глава 7), управление процессами (глава 8), взаимоотношения между различными процессами (глава 9) и сигналы (глава 10).
4. Дополнительно об операциях ввода-вывода: терминальный ввод-вывод (глава 11), расширенные операции ввода-вывода (глава 12) и процессы-демоны (глава 13).
5. IPC – взаимодействия между процессами (главы 14 и 15).
6. Примеры: библиотека базы данных (глава 16), взаимодействие с PostScript-принтером (глава 17), программа работы с модемом (глава 18) и использование псевдотерминалов (глава 19).

При чтении книги не лишним будет знание языка C, равно как и некоторый опыт использования UNIX. Изложение материала не предполагает наличия опыта разработки программ для UNIX. Книга предназначена для программистов, знакомых с UNIX или с другими операционными системами и желающих детально изучить возможности, предоставляемые большинством реализаций UNIX.

Примеры в книге

Данная книга содержит множество примеров – примерно 10 000 строк исходного кода. Все примеры написаны на языке C. Кроме того, примеры написаны в соответствии со стандартом ANSI C. Желательно, чтобы при чте-

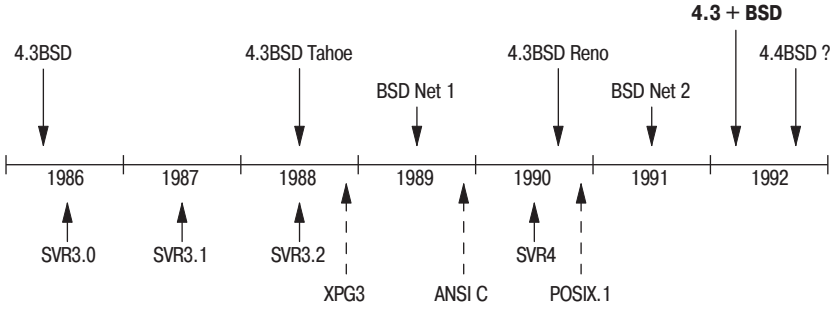
нии этой книги у вас под рукой была копия «Unix Programmer's Manual» (Руководства программиста UNIX) для вашей операционной системы, так как мы часто будем ссылаться на него при обсуждении малопонятных или зависящих от реализации особенностей.

Обсуждение практически каждой функции или системного вызова будет сопровождаться небольшой законченной программой. Это намного проще, чем рассматривать те же функции в больших программах, и позволит нам исследовать аргументы и возвращаемые значения. Так как некоторые из маленьких программ представляют собой достаточно искусственные примеры, мы включили в книгу несколько больших примеров (главы 16, 17, 18 и 19). Они демонстрируют решение задач, взятых из реальной жизни.

Все примеры программ были включены в текст книги прямо из файлов с исходными текстами. Копии всех примеров в виде файлов вы найдете на анонимном сервере FTP по адресу *ftp.uu.net*, в архиве *published/books/stevens.advprog.tar.Z*. Вы можете взять эти исходные тексты и экспериментировать с ними в вашей операционной системе.

Перечень систем, использовавшихся для тестирования примеров

К сожалению, все операционные системы находятся в постоянном движении. UNIX не является исключением. Следующая диаграмма показывает процесс эволюции различных версий System V и 4.xBSD.



Под обозначением 4.xBSD подразумеваются различные операционные системы от Computer Systems Research Group из Калифорнийского университета в Беркли. Эта группа также занимается распространением BSD Net 1 и BSD Net 2 – общедоступных исходные текстов операционных систем семейства 4.xBSD. Под обозначением SVRx подразумевается System V Release x от AT&T. XPG3 – это «X/Open Portability Guide, issue 3» (Руководство X/Open по обеспечению переносимости, выпуск 3). ANSI C – это стандарт ANSI языка программирования C. POSIX.1 – это стандарт ISO и IEEE на интерфейс UNIX-подобных операционных систем. Более подробно об этих стандартах и различных версиях UNIX мы поговорим в разделах 2.2 и 2.3.

В этой книге под обозначением *4.3+BSD* мы будем подразумевать версии UNIX, которые появились между выпусками BSD Net 2 и 4.4BSD.

К моменту написания книги версия 4.4BSD еще не была выпущена, поэтому было бы преждевременно говорить об операционной системе 4.4BSD. Однако эту серию операционных систем нужно было как-то обозначить, поэтому повсюду в книге мы будем использовать обозначение *4.3+BSD*.

Большинство примеров в книге были протестированы в четырех различных версиях UNIX:

1. UNIX System V/386 Release 4.0 Version 2.0 («чистая SVR4») от U.H. Corp. (UHC), работающая на процессоре Intel 80386.
2. 4.3+BSD от Computer Systems Research Group, отделение информатики, Калифорнийский университет в Беркли, работающая на рабочей станции от Hewlett Packard.
3. BSD/386 (производная от BSD Net 2) от Berkeley Software Design Inc., работающая на процессоре Intel 80386. Эта система практически полностью соответствует тому, что мы называем 4.3+BSD.
4. SunOS, версии 4.1.1 и 4.1.2 от Sun Microsystems (системы, в которых хорошо заметно наследие Беркли, но при этом много дополнений, пришедших из System V), работающие на SPARC-станциях SLC.

В книге представлены многочисленные тесты на производительность с указанием проверяемых операционных систем.

Благодарности

Я многим обязан моей семье за любовь, поддержку и множество выходных дней, потерянных за последние полтора года. Создание книги – во многом заслуга семьи. Спасибо вам, Салли, Билл, Эллен и Дэвид.

Я особенно благодарен Брайану Кернигану за его помощь при работе над книгой. Его многочисленные рецензии рукописи и ненавязчивые рекомендации по улучшению стиля изложения, надеюсь, будут заметны в окончательном варианте. Стив Раго также посвятил немало времени рецензированию рукописи и ответам на многие вопросы об устройстве и истории развития System V. Выражаю свою благодарность другим техническим рецензентам издательства Addison-Wesley, которые дали ценные комментарии по различным частям рукописи: Мори Баху (Maury Bach), Марку Эллису (Mark Ellis), Джефу Гитлину (Jeff Gitlin), Питеру Ханиману (Piter Honeyman), Джону Линдерману (John Linderman), Дугу Мак-Илрою (Doug McIlroy), Эви Немет (Evi Nemeth), Крэйгу Партриджу (Craig Partridge), Дейву Пресотто (Dave Presotto), Гэри Уилсону (Gary Wilson) и Гэри Райту (Gary Wright).

Кейт Бостик (Keith Bostic) и Кирк Мак-Кьюсик (Kirk McKusick) из U.C. Berkeley CSRG предоставили учетную запись, которая использовалась для проверки примеров на последней версии BSD (также большое спасибо Питеру Салусу (Peter Salus)). Сэм Нэйтарос (Sam Nataros) и Йоахим Саксен (Joachim Sacksen) из UHC предоставили копию операционной системы SVR4 для те-

стирования примеров. Трент Хейн (Trent Hein) помог получить альфа- и бета-версии BSD/386.

Другие мои друзья на протяжении последних лет часто оказывали мне немаловажную помощь: Пол Лукина (Paul Lucchina), Джо Годсил (Joe Godsil), Джим Хог (Jim Hogue), Эд Танкус (Ed Tankus) и Гэри Райт (Gary Wright). Редактор из издательства Addison-Wesley, Джон Уэйт (John Wait), был моим большим другом на протяжении всего этого времени. Он никогда не жаловался на срыв сроков и постоянное увеличение числа страниц. Отдельное спасибо Национальной оптической астрономической обсерватории (NOAO) и особенно Сиднею Уольфу (Sidney Wolff), Ричарду Уольфу (Richard Wolff) и Стиву Гранди (Steve Grandi) за предоставленное машинное время.

Настоящие книги о UNIX пишутся в формате troff, и данная книга также следует этой проверенной временем традиции. Копия книги, готовая к тиражированию, была подготовлена автором с помощью пакета groff, созданного Джеймсом Кларком (James Clark). Большое ему спасибо за такой замечательный пакет и за его быстрый отклик на замеченные ошибки. Быть может, мне когда-нибудь удастся разобраться с концевыми сносками в troff.

Я буду благодарен всем читателям, которые пришлют по электронной почте свои комментарии, предложения и замечания об ошибках.

Таксон, Аризона
Апрель 1992

Ричард Стивенс
rstevens@kohala.com
http://ww.kohala.com/~rstevens

15

Межпроцессное взаимодействие

15.1. Введение

В главе 8 мы описали примитивы управления процессами и увидели, как можно создать несколько процессов. Но единственный способ обмена информацией между этими процессами заключался в передаче открытых файловых дескрипторов через функции `fork` или `exec` или через файловую систему. Теперь мы рассмотрим другие способы взаимодействия процессов друг с другом – механизмы IPC (Interprocess Communication), или механизмы межпроцессного взаимодействия.

В прошлом механизмы IPC в UNIX представляли собой смесь самых разных концепций, лишь немногие из которых были переносимы между различными реализациями. Благодаря усилиям по стандартизации, предпринятым POSIX и The Open Group (ранее X/Open), ситуация значительно улучшилась, но некоторые различия все еще существуют. В табл. 15.1 приводится список различных форм IPC, которые поддерживаются всеми четырьмя платформами, обсуждаемыми в этой книге.

Обратите внимание: стандарт Single UNIX Specification (колонка SUS) разрешает реализациям поддерживать дуплексные неименованные каналы, но лишь поддержка полудуплексных неименованных каналов является обязательной. В реализациях, которые поддерживают дуплексные каналы, попрежнему корректно работают приложения, написанные для реализаций, поддерживающих только полудуплексные каналы. В табл. 15.1 мы использовали обозначение «дуплекс», чтобы выделить реализации, которые поддерживают полудуплексные каналы через использование дуплексных каналов.

В табл. 15.1 поддержка базовых функциональных возможностей обозначена точкой (•). Для случая с дуплексными каналами, если эта функциональность может предоставляться через сокеты домена UNIX (раздел 17.3), в соответствующих ячейках таблицы указана аббревиатура UDS (UNIX domain socket). Некоторое реализации поддерживают как базовую функциональ-

ность, так и сокеты домена UNIX, поэтому в этих ячейках указаны точка, и аббревиатура UDS вместе.

Как уже упоминалось в разделе 14.4, поддержка механизма STREAMS определяется стандартом Single UNIX Specification как необязательная. Именованные дуплексные каналы, которые реализуются на базе неименованных каналов STREAMS, также определяются стандартом Single UNIX Specification как необязательные для реализации. В ОС Linux поддержка STREAMS доступна в виде отдельного пакета LiS (от «Linux STREAMS»), который не устанавливается по умолчанию. Если та или иная функциональность реализована на базе дополнительных пакетов, которые не устанавливаются по умолчанию, в соответствующей ячейке мы указываем сокращение «доп.».

Таблица 15.1. Перечень механизмов IPC, доступных в UNIX

Тип IPC	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Полудуплексные неименованные каналы	•	дуплекс	•	•	дуплекс
Именованные каналы	•	•	•	•	•
Дуплексные неименованные каналы	Допускается	•, UDS	доп., UDS	UDS	•, UDS
Именованные дуплексные каналы	XSI, доп.	UDS	доп., UDS	UDS	•, UDS
Очереди сообщений	XSI	•	•		•
Семафоры	XSI	•	•	•	•
Разделяемая память	XSI	•	•	•	•
Сокеты	•	•	•	•	•
STREAMS	XSI, доп.		доп.		•

Первые семь видов IPC из табл. 15.1 обычно предназначены для организации взаимодействий между процессами, работающими на одной и той же машине. Последние два – сокеты и STREAMS – единственные формы IPC, которые повсеместно используются для организации взаимодействий между процессами, работающими на разных машинах, объединенных в сеть.

Мы разделили обсуждение механизмов межпроцессного взаимодействия на три главы. В этой главе мы будем рассматривать классические формы IPC: именованные и неименованные каналы, очереди сообщений, семафоры и разделяемую память. В следующей главе мы обсудим механизмы организации взаимодействий через сеть с помощью сокетов. И в главе 17 расскажем о некоторых расширенных возможностях IPC.

15.2. Неименованные каналы

Неименованные каналы (pipes, далее для краткости просто каналы) – это старейшая форма организации взаимодействий между процессами, предоставляемая операционными системами UNIX. Каналы имеют два ограничения:

1. Исторически они являются полудуплексными (то есть данные могут передаваться по ним только в одном направлении). Некоторые современные системы предоставляют дуплексные каналы, но для сохранения переносимости приложений никогда не следует пользоваться этой возможностью.
2. Каналы могут использоваться только для организации взаимодействия между процессами, которые имеют общего предка. Обычно канал создается родительским процессом, который затем вызывает функцию `fork`, после чего этот канал может использоваться для общения между родительским и дочерним процессами.

Далее (в разделе 15.5) мы увидим, что именованные каналы не имеют второго ограничения, а сокеты домена UNIX (unix domain sockets, раздел 17.3) и именованные каналы на базе STREAMS (раздел 17.2.2) – обоих ограничений.

Несмотря на указанные ограничения, полудуплексные каналы по-прежнему являются одной из наиболее широко используемых форм IPC. Каждый раз, когда вы вводите в командной строке последовательность команд, объединенных в конвейер, оболочка создает отдельный процесс для каждой команды и связывает с помощью канала стандартный вывод предыдущей команды со стандартным вводом следующей команды.

Неименованный канал создается с помощью функции `pipe`.

```
#include <unistd.h>

int pipe(int filedes[2]);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Через аргумент *filedes* возвращаются два файловых дескриптора: *filedes[0]* открыт для чтения, а *filedes[1]* – для записи. Данные, выводимые в *filedes[1]*, становятся входными данными для *filedes[0]*.

> В ОС 4.3BSD, 4.4BSD и Mac OS X 10.3 каналы реализованы с использованием сокетов домена UNIX. Даже несмотря на то, что сокеты по своей природе являются дуплексными, эти операционные системы ограничивают сокеты, используемые для организации каналов, таким образом, что они могут передавать информацию только в одном направлении.
>
> Стандарт POSIX.1 разрешает реализациям поддерживать дуплексные каналы. В таких реализациях дескрипторы *filedes[0]* и *filedes[1]* открываются как для чтения, так и для записи.

На рис. 15.1 изображены два примера использования полудуплексных каналов. Слева показан случай, когда канал обоими концами связан с одним и тем

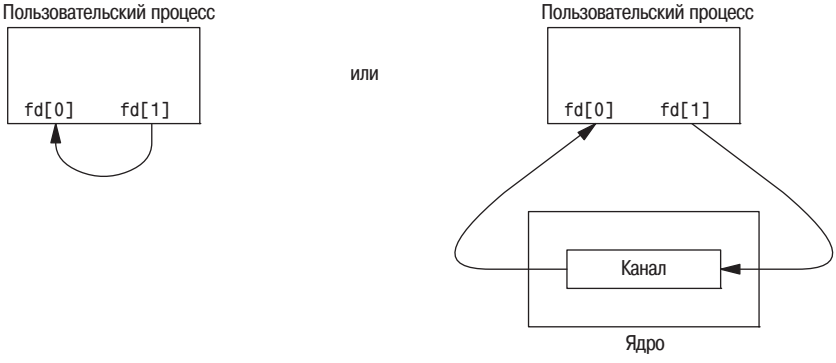


***Рис. 15.1.** Два примера использования полудуплексных каналов*

же процессом. Справа демонстрируется случай обмена данными между двумя процессами через ядро.

Функция `fstat` (раздел 4.2) для дескриптора любого конца канала возвращает тип файла FIFO. Убедиться в том, что дескриптор соответствует каналу, можно с помощью макроса `S_ISFIFO`.

> Стандарт POSIX.1 утверждает, что значение поля `st_size` структуры `stat` не определено для каналов. Но в большинстве систем вызов функции `fstat` для дескриптора, открытого на чтение, возвращает в поле `st_size` количество байт в канале, доступное для чтения. Однако это поведение не должно использоваться при разработке переносимых приложений.

Канал, который обоими концами связан с одним и тем же процессом, достаточно бесполезен. Обычно процесс, вызывающий функцию `pipe`, затем обращается к функции `fork`, создавая, таким образом, канал для передачи дан-

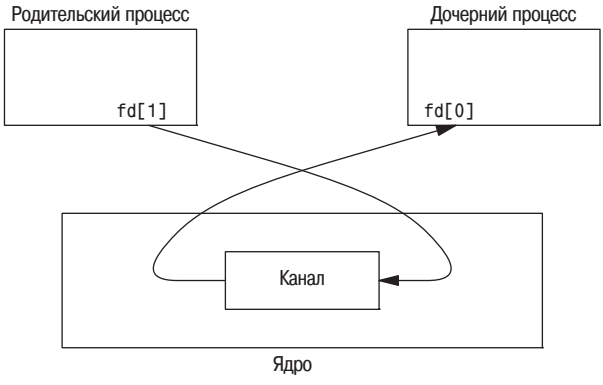


***Рис. 15.3.** Канал от родительского процесса к дочернему*

ных от родительского процесса к дочернему или наоборот. Этот сценарий показан на рис. 15.2.

Порядок действий, следующих за вызовом функции `fork`, зависит от того, в каком направлении мы желаем передавать данные. Если данные должны двигаться в направлении от родительского процесса к дочернему, тогда родитель закрывает дескриптор, открытый на чтение (`fd[0]`), а потомок закрывает дескриптор, открытый на запись (`fd[1]`). На рис. 15.3 показано окончательное состояние дескрипторов.

Чтобы организовать передачу в обратном направлении, родительский процесс закрывает `fd[1]`, а дочерний процесс – `fd[0]`.

Когда один из концов канала закрывается, в силу вступают следующие два правила.

1. Если попытаться прочитать данные из канала, в котором был закрыт дескриптор, открытый для записи, функция `read` вернет значение 0, чтобы сообщить о достижении конца файла после того, как все данные будут

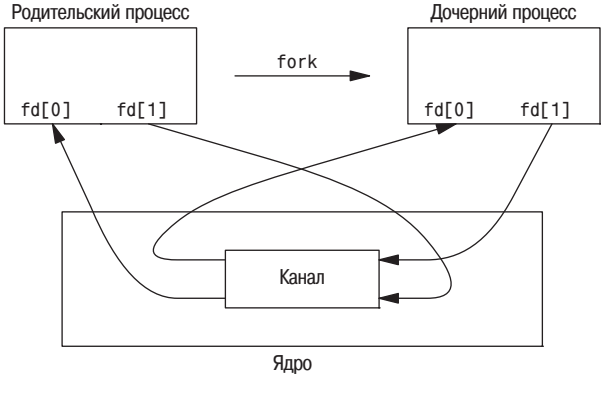


***Рис. 15.2.** Полудуплексные каналы после вызова функции fork*

прочитаны. (Технически, признак конца файла не будет сгенерирован до тех пор, пока не будут закрыты все дескрипторы, открытые для записи в канал. Такое возможно при создании дубликатов дескрипторов, благодаря чему сразу несколько процессов могут производить запись в канал. Однако обычно у канала имеется один дескриптор, открытый для чтения, и один дескриптор, открытый для записи. Когда в следующем разделе мы перейдем к изучению именованных каналов, то увидим, что зачастую в один именованный канал могут писать сразу несколько процессов.)

2. Если попытаться выполнить запись в канал, в котором был закрыт дескриптор, открытый для чтения, будет сгенерирован сигнал SIGPIPE. Если приложение игнорирует этот сигнал или перехватывает его и возвращает управление из обработчика сигнала нормальным образом, то функция write вернет значение –1 и код ошибки EPIPE в переменной errno.

При записи данных в канал размер буфера канала в ядре определяется константой PIPE_BUF. Если в канал записывается количество байт, не превышающее значения PIPE_BUF, то эти данные не будут перемежаться данными, записываемыми в канал (или FIFO) другими процессами. Если же мы попытаемся записать одним вызовом функции write большее, чем PIPE_BUF, количество байт, то записанные данные могут быть перемешаны с данными, поступившими от других процессов. Определить значение PIPE_BUF можно с помощью функции pathconf или fpathconf (раздел 2.11).

Пример

В листинге 15.1 представлена программа, которая создает канал между родительским и дочерним процессами и передает данные по этому каналу.

Листинг 15.1. Передача данных от родительского процесса к дочернему через канал

```
#include "apue.h"

int
main(void)
{
    int n;
    int fd[2];
    pid_t pid;
    char line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("ошибка вызова функции pipe");
    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid > 0) {              /* родительский процесс */
        close(fd[0]);
        write(fd[1], "привет, МИР\n", 12);
    } else {                           /* дочерний процесс */
```

```
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

В этом примере мы работали с дескрипторами канала напрямую, используя функции `write` и `read`. Но гораздо интереснее было бы продублировать тот или иной дескриптор на стандартный ввод или стандартный вывод. После этого дочерний процесс мог бы запустить некоторую программу, которая получает данные со стандартного ввода (из созданного нами канала) или производит запись на стандартный вывод (в канал).

Пример

Рассмотрим программу, которая должна выводить данные постранично. Вместо того, чтобы заново придумывать алгоритм постраничного вывода, который уже реализован некоторыми утилитами UNIX, мы попробуем воспользоваться программой постраничного просмотра, которую предпочитает пользователь. Чтобы избежать использования временного файла для хранения результатов и вызова функции `system` для отображения содержимого этого файла, мы воспользуемся каналом, по которому будем сразу же отправлять данные программе постраничного просмотра. Для этого мы сначала создадим канал, с помощью функции `fork` запустим дочерний процесс, переустановим дескриптор канала, открытый для чтения, на стандартный ввод и затем с помощью функции `exec` запустим программу постраничного просмотра. Листинг 15.2 показывает, как это можно сделать. (В этом примере программа принимает аргумент командной строки, определяющий имя файла, содержимое которого должно быть выведено. Но часто бывает, что данные, которые нужно вывести на терминал, уже находятся в памяти.)

Листинг 15.2. Передача файла программе постраничного просмотра

```
#include "apue.h"
#include <sys/wait.h>

#define DEF_PAGER "/bin/more" /* программа постраничного просмотра по умолчанию */

int
main(int argc, char *argv[])
{
    int n;
    int fd[2];
    pid_t pid;
    char *pager, *argv0;
    char line[MAXLINE];
    FILE *fp;

    if (argc != 2)
        err_quit("Использование: a.out <pathname>");

    if ((fp = fopen(argv[1], "r")) == NULL)
```

```
            err_sys("невозможно открыть %s", argv[1]);
        if (pipe(fd) < 0)
            err_sys("ошибка вызова функции pipe");

        if ((pid = fork()) < 0) {
            err_sys("ошибка вызова функции fork");
        } else if (pid > 0) {               /* родительский процесс */
            close(fd[0]);                   /* закрыть дескриптор для чтения */

            /* родительский процесс копирует argv[1] в канал */
            while (fgets(line, MAXLINE, fp) != NULL) {
                n = strlen(line);
                if (write(fd[1], line, n) != n)
                    err_sys("ошибка записи в канал");
            }
            if (ferror(fp))
                err_sys("ошибка вызова функции fgets");
            close(fd[1]);                   /* закрыть дескриптор для записи */
            if (waitpid(pid, NULL, 0) < 0)
                err_sys("ошибка вызова функции waitpid");
            exit(0);
        } else {                            /* дочерний процесс */
            close(fd[1]);                   /* закрыть дескриптор для записи */
            if (fd[0] != STDIN_FILENO) {
                if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
                    err_sys("ошибка переназначения канала на stdin");
                close(fd[0]);               /* уже не нужен после вызова dup2 */
            }
            /* определить аргументы для execl() */
            if ((pager = getenv("PAGER")) == NULL)
                pager = DEF_PAGER;
            if ((argv0 = strrchr(pager, '/')) != NULL)
                argv0++;                    /* перейти за последний слэш */
            else
                argv0 = pager;              /* в имени программы нет слэша */

            if (execl(pager, argv0, (char *)0) < 0)
                err_sys("ошибка запуска программы %s", pager);
        }
        exit(0);
    }
```

Перед вызовом функции `fork` создается канал. После вызова функции `fork` родительский процесс закрывает дескриптор канала, открытый для чтения, а дочерний процесс – дескриптор, открытый для записи. После этого дочерний процесс вызывает функцию `dup2`, с помощью которой переназначает конец канала, открытый для чтения, на стандартный ввод.

Когда мы дублируем один дескриптор в другой (`fd[0]` – на стандартный ввод в дочернем процессе), необходимо убедиться в том, что номер дескриптора не совпадает с тем, который нам нужен. Если бы это был дескриптор с нужным нам номером, то в результате вызова функций `dup2` и `close` единственная

копия дескриптора была бы закрыта. (Поведение функции dup2, когда оба ее аргумента равны, обсуждалось в разделе 3.12). В этой программе, если бы стандартный ввод не был открыт командной оболочкой, то функция fopen, вызываемая в самом начале, все равно открыла бы для файла дескриптор с номером 0, как наименьшим неиспользуемым номером дескриптора, – таким образом, fd[0] никогда не должен совпадать с дескриптором стандартного ввода. Однако всякий раз, когда мы обращаемся к функциям dup2 и close, дублируя один дескриптор в другой, в качестве меры предосторожности мы будем сначала сравнивать эти дескрипторы.

Обратите внимание, как мы использовали переменную окружения PAGER, чтобы получить имя программы постраничного вывода, предпочитаемой пользователем. Если таковая не определена, мы запускаем программу по умолчанию. Это наиболее распространенное правило использования переменных окружения.

Пример

Давайте вспомним функции TELL_WAIT, TELL_PARENT, TELL_CHILD, WAIT_PARENT и WAIT_CHILD из раздела 8.9. В листинге 10.17 была показана реализация этих функций на основе сигналов. В листинге 15.3 приводится реализация этих же функций, но уже на основе каналов.

Листинг 15.3. Процедуры синхронизации родительского и дочернего процессов

```
#include "apue.h"

static int pfd1[2], pfd2[2];

void
TELL_WAIT(void)
{
    if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
        err_sys("ошибка вызова функции pipe");
}

void
TELL_PARENT(pid_t pid)
{
    if (write(pfd2[1], "c", 1) != 1)
        err_sys("ошибка вызова функции write");
}

void
WAIT_PARENT(void)
{
    char c;

    if (read(pfd1[0], &c, 1) != 1)
        err_sys("ошибка вызова функции read");
    if (c != 'p')
        err_quit("WAIT_PARENT: получены некорректные данные");
}
```

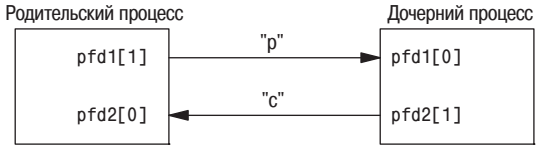


***Рис. 15.4.** Использование каналов для синхронизации родительского и дочернего процессов*

```
void
TELL_CHILD(pid_t pid)
{
    if (write(pfd1[1], "p", 1) != 1)
        err_sys("ошибка вызова функции write");
}

void
WAIT_CHILD(void)
{
    char c;

    if (read(pfd2[0], &c, 1) != 1)
        err_sys("ошибка вызова функции read");
    if (c != 'c')
        err_quit("WAIT_CHILD: получены некорректные данные");
}
```

Перед вызовом `fork` создаются два канала, как показано на рис. 15.4. Родительский процесс записывает в канал с помощью функции `TELL_CHILD` символ «`p`», а дочерний процесс с помощью функции `TELL_PARENT` записывает символ «`c`». Функции `WAIT_xxx` блокируются в системном вызове `read` до получения одиночного символа.

Обратите внимание, что в этой реализации каждый из каналов имеет два открытых для чтения дескриптора. Кроме того, что дочерний процесс может читать из дескриптора `pfd1[0]`, родительский процесс также имеет дескриптор, открытый для чтения. Но в данном случае это не имеет никакого значения, потому что родительский процесс не пытается читать из этого канала.

15.3. Функции popen и pclose

Поскольку чаще всего канал создается для взаимодействия с другим процессом, чтобы получать от него или отправлять ему данные, стандартная библиотека ввода-вывода традиционно поддерживает функции `pclose` и `popen`. Эти две функции берут на себя всю рутинную работу, которую мы до сих пор выполняли самостоятельно: создание канала, создание дочернего процесса,

закрытие неиспользуемых дескрипторов канала, запуск команды и ожидание завершения команды.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);

                                    Возвращает указатель на структуру FILE
                                    в случае успеха, NULL в случае ошибки

int pclose(FILE *fp);

                    Возвращает код завершения cmdstring, –1 в случае ошибки
```

Функция `popen` посредством функций `fork` и `exec` запускает на исполнение команду *`cmdstring`* и возвращает указатель на объект `FILE`. Если в аргументе *`type`* передается значение "`r`", указатель на файл будет связан со стандартным выводом *`cmdstring`* (рис. 15.5).

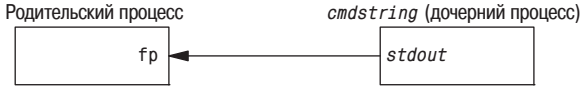


***Рис. 15.5.** Результат выполнения инструкции* `fp = popen(cmdstring, "r")`

Если в аргументе *`type`* передается значение "`w`", указатель на файл будет связан со стандартным вводом *`cmdstring`* (рис. 15.6).

Чтобы запомнить правила назначения второго аргумента функции `popen`, вспоминайте функцию `fopen`, которая возвращает объект `FILE`, открытый для чтения, если аргумент *`type`* имеет значение "`r`", и для записи, если аргумент *`type`* имеет значение "`w`".

Функция `pclose` закрывает поток ввода-вывода, ожидает завершения команды и возвращает код завершения командного интерпретатора, запущенного для выполнения команды *`cmdstring`*. (Код завершения рассматривался в разделе 8.6. Функция `system`, описанная в разделе 8.13, также возвращает код завершения.) Если командный интерпретатор не смог запуститься, функция `pclose` возвращает код завершения, как если бы командная оболочка вызвала функцию `exit(127)`.

Команда *`cmdstring`* запускается интерпретатором Bourne shell как

```
sh -c cmdstring
```

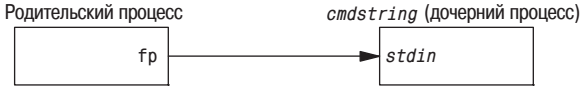


***Рис. 15.6.** Результат выполнения инструкции* `fp = popen(cmdstring, "w")`

Это означает, что командная оболочка производит интерпретацию всех специальных символов, которые встретятся в строке *cmdstring*. Это позволяет, например, выполнить команду

```
fp = popen("ls *.c", "r");
```

или

```
fp = popen("cmd 2>&1", "r");
```

Пример

Теперь попробуем переписать программу из листинга 15.2 таким образом, чтобы она использовала функцию popen. Текст новой программы приводится в листинге 15.4.

Листинг 15.4. Передача файла программе постраничного просмотра с использованием функции popen

```
#include "apue.h"
#include <sys/wait.h>

#define PAGER "${PAGER:-more}" /* либо значение переменной окружения, */
                               /* либо значение по умолчанию */
int
main(int argc, char *argv[])
{
    char line[MAXLINE];
    FILE *fpin, *fpout;

    if (argc != 2)
        err_quit("Использование: a.out <полное_имя_файла>");
    if ((fpin = fopen(argv[1], "r")) == NULL)
        err_sys("невозможно открыть %s", argv[1]);

    if ((fpout = popen(PAGER, "w")) == NULL)
        err_sys("ошибка вызова функции popen");

    /* передать argv[1] программе постраничного просмотра */
    while (fgets(line, MAXLINE, fpin) != NULL) {
        if (fputs(line, fpout) == EOF)
            err_sys("ошибка записи в канал");
    }
    if (ferror(fpin))
        err_sys("ошибка вызова функции fgets");
    if (pclose(fpout) == -1)
        err_sys("ошибка вызова функции pclose");
    exit(0);
}
```

Использование функции popen позволило значительно уменьшить размер программы.

Команда `${PAGER:-more}` **говорит о том, что следует использовать значение переменной окружения** `PAGER`**, если она определена и содержит непустую строку, в противном случае использовать строку** `more`**.**

Пример – функции popen и pclose

В листинге 15.5 приводится наша реализация функций `popen` **и** `pclose`**.**

Листинг 15.5. Функции popen и pclose

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>

/*
 * Указатель на массив, размещаемый во время выполнения.
 */
static pid_t *childpid = NULL;

/*
 * Будет получено из нашей функции open_max(), листинг 2.4.
 */
static int maxfd;

FILE *
popen(const char *cmdstring, const char *type)
{
    int i;
    int pfd[2];
    pid_t pid;
    FILE *fp;

    /* допустимы только "r" или "w" */
    if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0) {
        errno = EINVAL;         /* требование стандарта POSIX */
        return(NULL);
    }
    if (childpid == NULL) {     /* самый первый вызов функции */
        /* разместить массив идентификаторов потомков, заполненный нулями */
        maxfd = open_max();
        if ((childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
            return(NULL);
    }

    if (pipe(pfd) < 0)
        return(NULL);   /* значение errno будет установлено функцией pipe() */

    if ((pid = fork()) < 0) {
        return(NULL);   /* значение errno будет установлено функцией fork() */
    } else if (pid == 0) {  /* дочерний процесс */
        if (*type == 'r') {
            close(pfd[0]);
            if (pfd[1] != STDOUT_FILENO) {
```

```
                dup2(pfd[1], STDOUT_FILENO);
                close(pfd[1]);
            }
        } else {
            close(pfd[1]);
            if (pfd[0] != STDIN_FILENO) {
                dup2(pfd[0], STDIN_FILENO);
                close(pfd[0]);
            }
        }

        /* закрыть все дескрипторы в childpid[] */
        for (i = 0; i < maxfd; i++)
            if (childpid[i] > 0)
                close(i);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);
    }

    /* родительский процесс... */
    if (*type == 'r') {
        close(pfd[1]);
        if ((fp = fdopen(pfd[0], type)) == NULL)
            return(NULL);
    } else {
        close(pfd[0]);
        if ((fp = fdopen(pfd[1], type)) == NULL)
            return(NULL);
    }

    childpid[fileno(fp)] = pid; /* запомнить pid потомка для данного fd */
    return(fp);
}

int
pclose(FILE *fp)
{
    int fd, stat;
    pid_t pid;

    if (childpid == NULL) {
        errno = EINVAL;
        return(-1);             /* функция popen() никогда не вызывалась */
}

    fd = fileno(fp);
    if ((pid = childpid[fd]) == 0) {
        errno = EINVAL;
        return(-1);             /* fp не был открыт функцией popen() */
    }

    childpid[fd] = 0;
    if (fclose(fp) == EOF)
```

```
        return(-1);

    while (waitpid(pid, &stat, 0) < 0)
        if (errno != EINTR)
            return(-1); /* от waitpid получен код ошибки, отличный от EINTR */
    return(stat);              /* вернуть код завершения потомка */
}
```

В принципе функция popen похожа на тот код, который мы использовали ранее в этой главе, однако существует ряд моментов, которые необходимо принять во внимание. Прежде всего, каждый раз, когда вызывается функция popen, нужно запоминать идентификатор дочернего процесса и дескриптор файла либо указатель на объект FILE. Мы решили сохранять идентификатор дочернего процесса в массиве, который индексируется номерами файловых дескрипторов. Благодаря этому функция pclose, получая указатель на объект FILE, может восстановить по нему номер дескриптора файла с помощью функции fileno и затем извлечь из массива идентификатор дочернего процесса, чтобы передать его функции waitpid. Поскольку заданный процесс может вызывать функцию popen много раз, при первом обращении к функции popen мы размещаем в динамической памяти массив childpid максимального размера.

Вызовы функций pipe и fork и создание дубликатов дескрипторов для каждого процесса производятся практически так же, как мы это делали раньше.

Стандарт POSIX.1 требует, чтобы в дочернем процессе функция popen закрывала все потоки, которые были открыты предыдущими обращениями к ней. Для этого в дочернем процессе выполняется обход массива childpid и закрытие всех открытых дескрипторов.

Что случится, если процесс, вызывающий функцию pclose, установил обработчик сигнала SIGCHLD? В этом случае функция waitpid вернет код ошибки EINTR. Так как мы допускаем, что вызывающий процесс может перехватывать сигнал SIGCHLD (или любой другой сигнал, в результате чего может быть прервано выполнение системного вызова waitpid), то мы просто вызываем функцию waitpid еще раз, если ее выполнение было прервано в результате перехвата сигнала.

Обратите внимание: приложение может самостоятельно вызвать функцию waitpid и получить код завершения дочернего процесса, созданного функцией popen. В этом случае функция waitpid, вызываемая из pclose, обнаружит отсутствие дочернего процесса и вернет значение –1 с кодом ошибки ECHILD в переменной errno. Такое поведение регламентируется стандартом POSIX.1.

> Некоторые ранние версии pclose возвращали код ошибки EINTR, если выполнение функции wait было прервано перехваченным сигналом. Кроме того, в некоторых ранних версиях pclose игнорировались или блокировались сигналы SIGINT, SIGQUIT и SIGHUP. В стандарте POSIX.1 такое поведение считается недопустимым.

Обратите внимание, что функция popen никогда не должна вызываться из программ, для которых установлен бит set-user-ID или set-group-ID. Выполнение команды функцией popen эквивалентно выполнению инструкции

```
execl("/bin/sh", "sh", "-c", command, NULL);
```

которая запускает командный интерпретатор и команду *command* с окружением, унаследованным от вызывающей программы. В этом случае злоумышленник, манипулируя значениями переменных окружения, получает возможность запускать произвольные команды с повышенными привилегиями.

Функция popen особенно удобна для организации взаимодействия с простыми фильтрами, предназначенными для преобразования входных или выходных данных запускаемой команды. Это относится к случаям, когда программа сама выстраивает конвейер команд.

Пример

Рассмотрим пример программы, которая выводит на стандартный вывод приглашение и читает введенную строку со стандартного ввода. С помощью функции popen можно поместить некоторую программу между основным приложением и его стандартным вводом, чтобы выполнить первичную обработку входных данных. На рис. 15.7 показано, как взаимодействуют процессы в такой ситуации.

В качестве первичной обработки может выполняться, например, автоматическое дополнение имен файлов или предоставление истории команд (сохранение ранее введенных команд).

В листинге 15.6 показан пример подобного простого фильтра. Этот фильтр копирует данные со стандартного ввода на стандартный вывод, преобразуя символы верхнего регистра в нижний регистр. В следующем разделе, когда мы будем рассказывать о сопроцессах, вы узнаете, почему мы вставили вызов функции fflush после вывода символа перевода строки.

Листинг 15.6. Фильтр для преобразования символов верхнего регистра в нижний регистр

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

int
main(void)
{
```

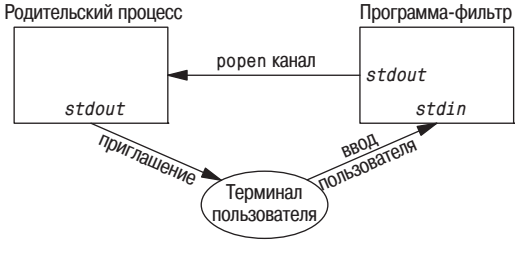


***Рис. 15.7.** Первичная обработка входных данных с помощью функции popen*

```
    int c;

    while ((c = getchar()) != EOF) {
        if (isupper(c))
            c = tolower(c);
        if (putchar(c) == EOF)
            err_sys("ошибка вывода символа");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}
```

Мы скомпилировали этот фильтр в исполняемый файл `myuclc`, который будет вызываться функцией `popen` из программы, представленной в листинге 15.7.

Листинг 15.7. Вызов фильтра преобразования регистра символов при чтении данных

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char line[MAXLINE];
    FILE *fpin;

    if ((fpin = popen("myuclc", "r")) == NULL)
        err_sys("ошибка вызова функции popen");

    for ( ; ; ) {
        fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL)   /* чтение из канала */
            break;
        if (fputs(line, stdout) == EOF)
            err_sys("ошибка вызова функции fputs");
    }
    if (pclose(fpin) == -1)
        err_sys("ошибка вызова функции pclose");
    putchar('\n');
    exit(0);
}
```

Вызов `fflush` после вывода строки приглашения необходим по той простой причине, что стандартный вывод обычно буферизуется построчно, а строка приглашения не включает символ перевода строки.

15.4. Сопроцессы

В системе UNIX фильтрами называются программы, которые читают входные данные со стандартного ввода и выводят результаты на стандартный вы-

вод. Как правило, фильтры используются при конвейерной обработке данных. Фильтр является *сопроцессом*, если одна и та же программа предоставляет данные для фильтра и получает его вывод.

Командная оболочка Korn shell предоставляет возможность запуска сопроцессов (см. [Bolsky and Korn 1995]). Командные оболочки Bourne shell, Bourne-again shell и C shell такой возможности не имеют. Обычно сопроцесс запускается из командной оболочки в фоновом режиме, и его стандартный ввод и стандартный вывод соединены с другой программой посредством каналов. Несмотря на то, что синтаксис команд оболочки, необходимых для запуска сопроцесса и соединения его ввода и вывода с другим процессом, весьма запутан (за подробностями обращайтесь к [Bolsky and Korn 1995], стр. 62–63), работа с сопроцессами достаточно удобна из программ, написанных на C.

Учитывая однонаправленную природу каналов, для организации взаимодействия с сопроцессом необходимо создать два однонаправленных канала – один к стандартному вводу сопроцесса и другой от его стандартного вывода. После этого можно записать данные на стандартный ввод сопроцесса, обработать их и прочитать результат с его стандартного вывода.

Пример

Давайте рассмотрим пример сопроцесса. Основной процесс создает два канала: один связан со стандартным вводом сопроцесса, а второй – с его стандартным выводом. Эта ситуация показана на рис. 15.8.

Программа из листинга 15.8 представляет собой простейший сопроцесс, который принимает два числа со стандартного ввода, вычисляет сумму и выводит ее на стандартный вывод. (Сопроцессам обычно доверяют более серьезные задачи. Это во многом искусственный пример, он придуман лишь с целью изучить механизмы взаимодействия между процессами.)

Листинг 15.8. Простейший фильтр, который складывает два числа

```
#include "apue.h"

int
main(void)
{
    int n, int1, int2;
    char line[MAXLINE];

    while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
        line[n] = 0;      /* завершить строку нулевым символом */
```

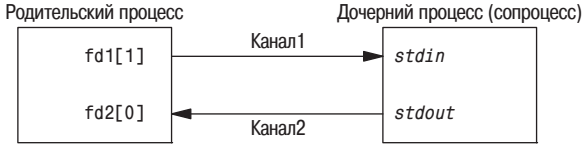


***Рис. 15.8.** Запись на стандартный ввод и чтение со стандартного вывода сопроцесса*

```
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            sprintf(line, "%d\n", int1 + int2);
            n = strlen(line);
            if (write(STDOUT_FILENO, line, n) != n)
                err_sys("ошибка вызова функции write");
        } else {
            if (write(STDOUT_FILENO, "неверные аргументы\n", 19) != 19)
                err_sys("ошибка вызова функции write");
        }
    }
    exit(0);
}
```

Мы скомпилировали эту программу в исполняемый файл `add2`**.**

Программа, представленная листингом 15.9, читает два числа со стандартного ввода и вызывает сопроцесс `add2`**. Значение, полученное от сопроцесса, выводится на стандартный вывод.**

Листинг 15.9. Программа, которая использует фильтр add2

```
#include "apue.h"

static void sig_pipe(int);              /* обработчик сигнала */

int
main(void)
{
    int n, fd1[2], fd2[2];
    pid_t pid;
    char line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("ошибка вызова функции signal");

    if (pipe(fd1) < 0 || pipe(fd2) < 0)
        err_sys("ошибка вызова функции pipe");

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid > 0) {               /* родительский процесс */
        close(fd1[0]);
        close(fd2[1]);
        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd1[1], line, n) != n)
                err_sys("ошибка записи в канал");
            if ((n = read(fd2[0], line, MAXLINE)) < 0)
                err_sys("ошибка чтения из канала");
            if (n == 0) {
                err_msg("канал был закрыт в дочернем процессе");
                break;
            }
            line[n] = 0; /* добавить завершающий нулевой символ */
            if (fputs(line, stdout) == EOF)
```

```
                err_sys("ошибка вызова функции fputs");
        }
        if (ferror(stdin))
            err_sys("ошибка получения данных со стандартного ввода");
        exit(0);
    } else {                            /* дочерний процесс */
        close(fd1[1]);
        close(fd2[0]);
        if (fd1[0] != STDIN_FILENO) {
            if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("ошибка вызова функции dup2 для stdin");
            close(fd1[0]);
        }
        if (fd2[1] != STDOUT_FILENO) {
            if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("ошибка вызова функции dup2 для stdout");
            close(fd2[1]);
        }
        if (execl("./add2", "add2", (char *)0) < 0)
            err_sys("ошибка вызова функции execl");
    }
    exit(0);
}

static void
sig_pipe(int signo)
{
    printf("перехвачен сигнал SIGPIPE\n");
    exit(1);
}
```

Эта программа создает два канала, дочерний и родительский процессы закрывают ненужные дескрипторы каналов. Мы должны использовать два канала: один в качестве стандартного ввода сопроцесса, а второй в качестве его стандартного вывода. Затем, перед вызовом `execl`, дочерний процесс вызывает функцию `dup2`, чтобы перенести дескрипторы каналов на свои стандартные устройства ввода и вывода.

Если скомпилировать и запустить программу из листинга 15.9, она будет работать так, как мы и ожидали. Если в то время, когда основная программа ждет ввода двух чисел, завершить сопроцесс `add2` с помощью команды `kill`, то при попытке выполнить запись в канал, для которого отсутствует читающий процесс, основная программа получит сигнал `SIGPIPE` (упражнение 15.4).

В табл. 15.1 было указано, что функция `pipe` не во всех системах создает дуплексные каналы. В листинге 17.1 приводится еще одна версия этого примера. Вместо двух полудуплексных каналов она использует один дуплексный канал в тех системах, которые поддерживают дуплексные неименованные каналы.

Пример

В примере сопроцесса add2 (листинг 15.8) мы преднамеренно использовали низкоуровневые операции ввода-вывода (системные вызовы UNIX): read и write. А может ли сопроцесс использовать стандартную библиотеку ввода-вывода? Такая версия сопроцесса приводится в листинге 15.10.

Листинг 15.10. Простейший фильтр, складывающий два числа и реализованный с применением стандартной библиотеки ввода-вывода

```
#include "apue.h"

int
main(void)
{
    int int1, int2;
    char line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL) {
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            if (printf("%d\n", int1 + int2) == EOF)
                err_sys("ошибка вызова функции printf");
        } else {
            if (printf("неверные аргументы\n") == EOF)
                err_sys("ошибка вызова функции printf");
        }
    }
    exit(0);
}
```

Если теперь попытаться вызвать этот новый сопроцесс из программы, представленной листингом 15.9, она перестанет работать. Проблема в том, что стандартная библиотека по умолчанию буферизует операции ввода-вывода. Когда вызывается программа из листинга 15.10, при первом обращении к функции fgets стандартная библиотека ввода-вывода размещает буфер и выбирает режим буферизации. Поскольку стандартный ввод является каналом, библиотека по умолчанию выбирает режим полной буферизации. То же самое происходит и со стандартным выводом. Пока программа add2 ожидает поступления данных со стандартного ввода, основная программа (из листинга 15.9) ожидает поступления данных из канала. В результате возникает тупиковая ситуация.

Можно изменить программу из листинга 15.10, добавив перед циклом while следующие четыре строки:

```
if (setvbuf(stdin, NULL, _IOLBF, 0) != 0)
    err_sys("ошибка вызова функции setvbuf");
if (setvbuf(stdout, NULL, _IOLBF, 0) != 0)
    err_sys("ошибка вызова функции setvbuf");
```

Эти строки заставят функцию fgets вернуть управление, когда строка символов будет записана в канал, а функцию printf – вызвать fflush после вывода символа перевода строки (подробное описание режимов буферизации

в стандартной библиотеке ввода-вывода приводилось в разделе 5.4). Добавив явные вызовы функций setvbuf, мы исправим ошибку в программе из листинга 15.10.

Иная методика требуется, если у нас нет возможности изменить программу, к стандартному выводу которой мы присоединяем канал. Например, при использовании в нашей программе в качестве сопроцесса программы awk(1) (вместо программы add2), следующий код не будет работать:

```
#! /bin/awk -f
{ print $1 + $2 }
```

Причина опять кроется в буферизации операций ввода-вывода. Но на этот раз мы не можем изменить программу awk (если, конечно, мы не имеем исходных текстов этой программы). У нас нет возможности внести изменения в исполняемый файл, чтобы изменить режим буферизации по умолчанию.

Решение этой проблемы заключается в том, чтобы заставить сопроцесс (в данном случае awk) думать, что его стандартный ввод и стандартный вывод соединены с терминалом. Это заставит функции стандартной библиотеки ввода-вывода в сопроцессе установить режим построчной буферизации для двух потоков ввода-вывода, как если бы функция setvbuf была вызвана явным образом. Для этого в главе 19 мы будем использовать псевдотерминалы.

15.5. FIFO

Каналы FIFO (First In First Out – первым пришел, первым ушел) иногда еще называют именованными каналами. Неименованные каналы могут использоваться только для организации взаимодействия процессов, которые имеют общего предка, создавшего каналы. (Исключение из этого правила составляют монтируемые каналы на основе механизма STREAMS, которые мы будем рассматривать в разделе 17.2.2.) С помощью каналов FIFO можно организовать взаимодействие между процессами, которые не связаны «родственными узами».

В главе 4 мы уже видели, что FIFO – это особый тип файлов. Определенный код в поле st_mode структуры stat (раздел 4.2) указывает, что файл является каналом FIFO. Проверить файл на принадлежность к типу FIFO можно с помощью макроса S_ISFIFO.

Создание канала FIFO очень похоже на создание обычного файла. Канал с полным именем *pathname* действительно создается в пределах файловой системы.

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

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

Возвращает 0 в случае успеха, –1 в случае ошибки

Аргумент *mode* для функции mkfifo указывается точно так же, как и для функции open (раздел 3.3). Правила назначения пользователя и группы владельца FIFO совпадают с описанными в разделе 4.6.

После создания канала FIFO с помощью функции mkfifo мы можем открыть его функцией open. Все обычные функции ввода-вывода (close, read, write, unlink и др.) могут работать с каналами FIFO.

> Приложения могут создавать каналы FIFO с помощью функции mknod. Поскольку изначально стандарт POSIX.1 не включал в себя функцию mknod, специально для этого стандарта была придумана функция mkfifo. Сейчас функция mknod включена в стандарт в виде расширения XSI. В большинстве систем функция mkfifo создает канал FIFO с помощью функции mknod.
>
> Стандарт POSIX.1 также включает команду mkfifo(1). Все четыре платформы, обсуждаемые в этой книге, поддерживают данную команду. Она позволяет создавать каналы FIFO из командной оболочки, чтобы затем использовать их для перенаправления ввода-вывода.

При открытии FIFO наличие флага O_NONBLOCK оказывает следующее влияние.

- В обычной ситуации (флаг O_NONBLOCK не указан) операция открытия FIFO только для чтения будет заблокирована до тех пор, пока другой процесс не откроет канал для записи. Аналогично, операция открытия только для записи будет заблокирована, пока другой процесс не откроет канал для чтения.
- Если флаг O_NONBLOCK указан, при попытке открыть канал только для чтения функция open сразу же вернет управление. Но при попытке открыть канал только для записи функция open вернет значение –1 и код ошибки ENXIO в переменной errno, если канал не был открыт другим процессом для чтения.

Как и в случае с неименованными каналами, если мы попытаемся выполнить запись в FIFO, который не был открыт для чтения, процесс получит сигнал SIGPIPE. Когда последний пишущий в FIFO процесс закроет канал, читающий процесс получит признак конца файла.

Нередко запись данных в канал FIFO выполняется из нескольких процессов. Это означает, что необходимо побеспокоиться об атомарности операции записи, если мы хотим избежать смешивания данных, поступающих от разных процессов. (Решение этой проблемы мы увидим в разделе 17.2.2.) Максимальный объем данных, который может быть атомарно записан в канал FIFO, определяется, как и для неименованных каналов, константой PIPE_BUF.

Каналы FIFO применяются в двух случаях:

1. Каналы FIFO используются командными оболочками для передачи данных от одного конвейера команд другому без создания временных файлов для хранения промежуточных данных.
2. Каналы FIFO используются для организации взаимодействий типа клиент-сервер.

Каждый из этих случаев мы рассмотрим на конкретных примерах.

Пример – использование FIFO для дублирования вывода

Каналы FIFO могут использоваться для дублирования данных, передаваемых между сериями команд оболочки. Это помогает избежать создания временных файлов для хранения промежуточных данных (как и неименованные каналы). Но если неименованные каналы могут служить исключительно для линейного объединения процессов в конвейер, то каналы FIFO, благодаря наличию имени, могут использоваться для нелинейного объединения.

Рассмотрим ситуацию, когда необходимо обработать отфильтрованные данные дважды. Эта ситуация показана на рис. 15.9.

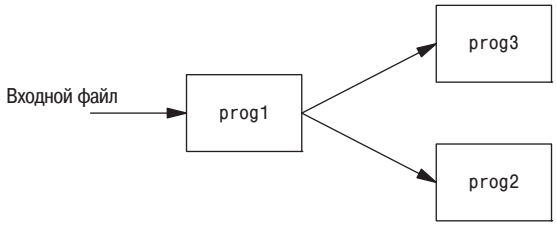


***Рис. 15.9.** Ситуация, когда необходимо обработать отфильтрованные данные дважды*

С помощью канала FIFO и программы `tee`(1) можно реализовать эту процедуру без использования временного файла. (Программа `tee` копирует данные, получаемые со стандартного ввода, на стандартный вывод и в файл, заданный в командной строке.)

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

Эта последовательность команд создает канал FIFO, после чего запускает в фоновом режиме программу `prog3`, которая читает данные из канала. Затем запускается фильтр `prog1`, вывод которого через команду `tee` поступает в канал и на вход программы `prog2`. На рис. 15.10 показана схема взаимодействия процессов в этой ситуации.

Пример – взаимодействия типа клиент-сервер

Еще одна область применения каналов FIFO – передача данных между клиентом и сервером. Если у нас имеется сервер, который обслуживает многочисленные клиентские приложения, каждый клиент может посылать запросы через канал FIFO с известным именем, заранее созданный сервером. (Имеется в виду, что полное имя канала FIFO заранее известно всем клиентам, которые нуждаются в услугах сервера.) На рис. 15.11 показана схема такого взаимодействия. Поскольку в этой ситуации писать в канал FIFO могут сразу несколько процессов, необходимо, чтобы размеры запросов не пре-

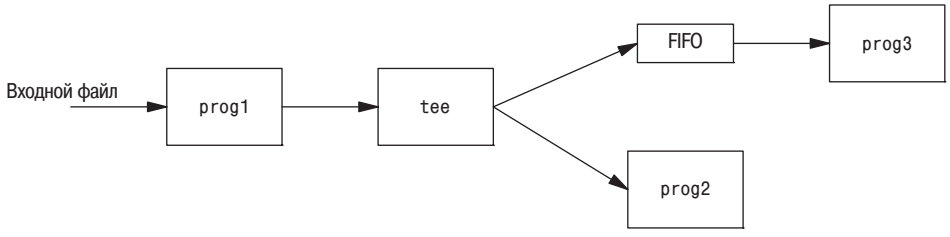


***Рис. 15.10.** Использование канала FIFO и команды tee для обработки потока данных двумя процессами*

вышали величины PIPE_BUF. Это позволит избежать смешивания данных, записываемых различными процессами.

При использовании каналов FIFO для организации взаимодействий такого типа возникает проблема с отправкой ответа сервера клиенту. Для этого не может использоваться единственный канал FIFO, поскольку клиенты не смогут отличить ответ сервера на свой запрос от ответов на запросы других клиентов. Одно из решений состоит в том, чтобы каждый клиент отсылал вместе с запросом идентификатор процесса. Тогда сервер мог бы создавать каналы FIFO для связи с каждым клиентом, генерируя имя канала на основе идентификатора процесса клиента. Например, сервер может создавать каналы FIFO с именами /tmp/serv1.XXXXX, где XXXXX – идентификатор процесса клиента. На рис. 15.12 показана схема такого взаимодействия.

Такая схема вполне работоспособна, хотя сервер не может обнаружить ситуацию аварийного завершения клиента. Это приводит к тому, что каналы FIFO, созданные для взаимодействия с конкретными клиентами, остаются в файловой системе. Кроме того, сервер должен предусматривать обработку сигнала SIGPIPE, поскольку клиент может послать запрос и завершить работу, не дожи-

***Рис. 15.11.** Обмен данными между клиентами и сервером с помощью канала FIFO*

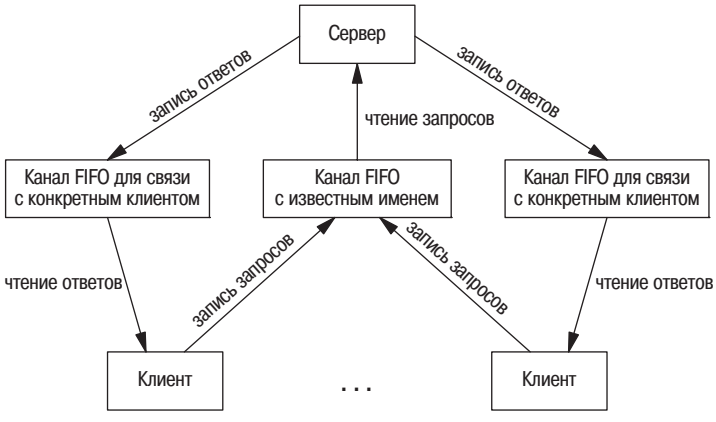


***Рис. 15.12.** Организация взаимодействий типа клиент-сервер с помощью каналов FIFO*

даясь получения ответа и оставляя свой канал FIFO с одним пишущим процессом (сервер), но без читающего процесса. Более элегантное решение этой проблемы мы увидим в разделе 17.2.2, когда будем обсуждать монтируемые каналы STREAMS.

В ситуации, показанной на рис. 15.12, если сервер откроет канал FIFO с заранее предопределенным именем только для чтения, то всякий раз, когда количество клиентов будет достигать 0, сервер будет получать из канала признак конца файла. Чтобы этого не происходило, сервер обычно открывает канал FIFO как для чтения, так и записи (упражнение 15.10).

15.6. XSI IPC

Три типа IPC, которые называются XSI IPC, – очереди сообщений, семафоры и разделяемая память – имеют много общего. В этом разделе мы рассмотрим общие для всех трех типов взаимодействий характеристики, а в следующем разделе – функции, специфичные для каждого из них.

> Функции XSI IPC целиком основаны на функциях System V IPC. Эти три типа взаимодействий появились в 70-х годах во внутренней версии UNIX AT&T, которая называлась Columbus UNIX. Позднее эти механизмы IPC были добавлены в System V. Их часто критикуют за то, что они используют свою собственную схему именования объектов, а не файловую систему.
>
> В табл. 15.1 было указано, что очереди сообщений, семафоры и разделяемая память определяются стандартом Single UNIX Specification как расширения XSI.

15.6.1. Идентификаторы и ключи

Каждой структуре IPC (очереди сообщений, семафору или сегменту разделяемой памяти) в ядре соответствует неотрицательное целое число – *идентификатор*. Например, чтобы отправить сообщение в очередь или получить его,

достаточно знать идентификатор очереди. В отличие от дескрипторов файлов, идентификаторы IPC не являются маленькими целыми числами. Каждый раз, когда создается какая-либо структура IPC, идентификатор, присваиваемый этой структуре, все время увеличивается, пока не достигнет максимально возможного значения для целых чисел, после чего сбрасывается в ноль.

Идентификатор – это внутреннее имя объекта IPC. Процессам же необходим механизм внешних имен, чтобы организовать взаимодействие через определенный объект IPC. Для этого каждому объекту IPC ставится в соответствие *ключ*, который и выступает в роли внешнего имени.

Всякий раз, когда создается структура IPC (`msgget`, `semget` или `shmget`), обязательно должен быть указан ключ. Тип данных ключа является одним из основных системных типов данных – `key_t`, который часто определяется в заголовочном файле `<sys/types.h>` как длинное целое со знаком. Ядро выполняет преобразование этого ключа во внутренний идентификатор.

Существуют разные способы организовать взаимодействие между клиентом и сервером через структуру IPC.

1. Сервер может создать новую структуру IPC с ключом `IPC_PRIVATE` и записать куда-нибудь (например, в файл) полученный идентификатор, чтобы сделать его доступным для клиента. Ключ `IPC_PRIVATE` гарантирует, что сервер создаст совершенно новую структуру IPC. Недостаток этого метода заключается в том, что в нем применяются операции с файловой системой – сервер должен записать идентификатор в файл, а клиент должен прочитать его из файла.

   Ключ `IPC_PRIVATE` также может использоваться для организации взаимодействия родительского и дочернего процессов. Родительский процесс создает новую структуру IPC с ключом `IPC_PRIVATE`, а полученный в результате идентификатор останется доступным для потомка после вызова функции `fork`. Дочерний процесс может передать полученный идентификатор новой программе в качестве аргумента функции `exec`.

2. Клиент и сервер могут договориться об использовании предопределенного ключа, задав его, например, в общем заголовочном файле. После этого сервер может создавать структуру IPC с заданным ключом. Но такое решение также не лишено недостатков – есть вероятность, что в системе уже существует некоторая структура IPC с точно таким же ключом. В этом случае функции создания структуры (`msgget`, `semget` или `shmget`) будут возвращать признак ошибки. Сервер должен правильно обработать ошибочную ситуацию, удалить существующую структуру IPC и попытаться создать ее снова.

3. Клиент и сервер могут договориться об использовании некоторого полного имени файла и идентификатора проекта (идентификатор проекта – это символ с кодом от 0 до 255), на основе которых с помощью функции `ftok`

можно получить значение ключа. После этого полученный ключ может использоваться точно так же, как и в предыдущем случае.

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

key_t ftok(const char *path, int id);
```

Возвращает ключ в случае успеха, (key_t) −1 в случае ошибки

Аргумент *path* должен содержать имя существующего файла. В создании ключа используются только 8 младших бит аргумента *id*.

Обычно при создании ключа используются значения полей st_dev и st_ino структуры stat (раздел 4.2), соответствующей файлу с заданным именем, в комбинации с идентификатором проекта. Если имена файлов различаются, то функция ftok обычно возвращает разные ключи. Однако, поскольку и номера индексных узлов, и ключи часто хранятся в виде длинных целых чисел со знаком, то при создании ключа может происходить некоторая потеря информации. Это означает, что существует вероятность, когда при использовании различных имен файлов будут сгенерированы одинаковые ключи, если в обоих случаях использовались одинаковые значения идентификатора проекта.

Все три функции get (msgget, semget и shmget) имеют одинаковые аргументы: *key* и *flag*. Новая структура IPC создается (обычно сервером) в том случае, если в аргументе *key* передается значение IPC_PRIVATE либо заданный ключ не соответствует какой-либо существующей структуре IPC данного типа, и в аргументе *flag* передается флаг IPC_CREAT. Чтобы получить ссылку на существующую очередь (обычно со стороны клиента), в аргументе *key* нужно передать значение, совпадающее с ключом, использовавшимся при создании этой очереди, а флаг IPC_CREAT не должен быть установлен.

Обратите внимание: если используется ключ IPC_PRIVATE, то получить ссылку на существующую очередь невозможно, так как с помощью этого специального ключа всегда создается новая очередь. Чтобы иметь возможность обращаться к существующей очереди, которая была создана с ключом IPC_PRIVATE, мы должны узнать связанный с ней идентификатор и затем использовать его во всех остальных функциях работы с объектом IPC (таких как msgsnd или msgrcv) в обход функции get.

Если нужно создать новую структуру IPC, а не получить ссылку на существующую, мы должны в аргументе *flag* вместе с флагом IPC_CREAT указать флаг IPC_EXCL. В результате, если данная структура IPC уже существует, функция вернет признак ошибки с кодом EEXIST. (Очень напоминает правила определения флагов O_CREAT и O_EXCL функции open.)

15.6.2. Структура прав доступа

С каждой структурой IPC механизм XSI IPC связывает структуру ipc_perm. Эта структура определяет права доступа к объекту и его владельца. Она содержит как минимум следующие поля:

```
struct ipc_perm {
    uid_t uid;     /* эффективный идентификатор пользователя владельца */
    gid_t gid;     /* эффективный идентификатор группы владельца */
    uid_t cuid;    /* эффективный идентификатор пользователя создателя */
    gid_t cgid;    /* эффективный идентификатор группы создателя */
    mode_t mode;   /* режим доступа */
    .
    .
    .
};
```

Каждая реализация включает в эту структуру дополнительные поля. Полное определение этой структуры в своей системе вы найдете в заголовочном файле <sys/ipc.h>.

Все поля структуры инициализируются при создании структуры IPC. Позднее можно изменить состояние полей uid, gid и mode с помощью функций msgctl, semctl или shmctl. Чтобы иметь возможность изменять эти значения, процесс должен обладать правами создателя структуры или правами суперпользователя. Изменение этих полей структуры похоже на вызов функций chown или chmod для обычного файла.

Значения поля mode напоминают значения, которые мы уже видели в табл. 4.5, за исключением права на исполнение. Кроме того, применительно к очередям сообщений и разделяемой памяти используются термины «право на чтение» и «право на запись», а применительно к семафорам – «право на чтение» и «право на изменение». В табл. 15.2 приводится список различных прав доступа к каждой из структур IPC.

Таблица 15.2. Права доступа XSI IPC

Право доступа	Бит
user-read – доступно пользователю для чтения	0400
user-write – доступно пользователю для записи (изменения)	0200
group-read – доступно группе для чтения	0040
group-write – доступно группе для записи (изменения)	0020
other-read – доступно остальным для чтения	0004
other-write – доступно остальным для записи (изменения)	0002

Некоторые реализации определяют символические константы для каждого бита прав доступа, однако имена этих констант не стандартизированы в Single UNIX Specification.

15.6.3. Конфигурируемые пределы

Мы можем столкнуться с встроенными пределами для всех трех форм XSI IPC. Большинство из них могут быть изменены при переконфигурировании ядра. Мы будем рассматривать эти пределы при обсуждении каждой из трех форм IPC.

Каждая из платформ предоставляет свой собственный способ получения и изменения конкретных пределов. В ОС FreeBSD 5.2.1, Linux 2.4.22 и Mac OS X 10.3 имеется команда `sysctl`, с помощью которой можно просмотреть и изменить конфигурационные параметры ядра. В Solaris 9 для изменения конфигурационных параметров ядра нужно отредактировать файл `/etc/system` и перезагрузить систему.

В Linux можно просмотреть пределы, связанные с IPC, запустив команду `ipcs -l`. В FreeBSD ей соответствует команда `ipcs -T`. В Solaris можно просмотреть значения настраиваемых параметров, запустив команду `sysdef -i`.

15.6.4. Преимущества и недостатки

Фундаментальная проблема всех форм XSI IPC заключается в том, что структуры IPC привязаны к системе в целом, а не к конкретному процессу, и не имеют счетчика ссылок. Например, если мы создадим очередь сообщений, поместим в нее некоторое сообщение и завершим процесс, то очередь не будет удалена. Сообщение останется в системе до тех пор, пока не будет прочитано каким либо процессом с помощью функции `msgrcv`, удалено с помощью функции `msgctl` или команды `ipcrm`(1), или пока система не будет перезагружена. Сравните это с неименованными каналами, которые удаляются автоматически, когда завершается последний процесс, имеющий ссылку на этот канал. В случае с FIFO имя канала остается в системе до тех пор, пока явно не будет удалено, но данные удаляются из канала FIFO автоматически, когда завершается последний процесс, имеющий ссылку на этот канал.

Другая проблема, связанная с механизмами XSI IPC, заключается в том, что они не имеют имен в файловой системе. Мы не можем получить к ним доступ или изменить их свойства с помощью функций, описанных в главах 3 и 4. Для поддержки этих объектов IPC в ядро была добавлена почти дюжина новых системных вызовов (`msgget`, `semop`, `shmat` и другие). Мы не можем получить список существующих объектов IPC с помощью команды `ls`, удалить их с помощью команды `rm` и изменить права доступа к ним с помощью команды `chmod`. Вместо них должны использоваться две новые команды – `ipcs`(1) и `ipcrm`(1).

Так как эти формы IPC не используют файловые дескрипторы, нельзя использовать для работы с ними функции мультиплексирования ввода-вывода (`select` и `poll`). Это осложняет одновременную работу с более чем одной структурой IPC или использование какой-либо из этих структур совместно с файлами или устройствами ввода-вывода. Например, мы не можем организовать на стороне сервера ожидание сообщения, которое может поступить по одной из двух очередей, не применяя для этого ту или иную форму цикла активного ожидания.

Краткий обзор системы диалоговой обработки запросов, построенной на основе System V IPC, приводится в [Andrade, Carges, and Kovach 1989]. Авторы этой книги утверждают, что принцип именования, используемый System V IPC (идентификаторы), является преимуществом, а не недостатком, как мы говорили выше, потому что идентификаторы позволяют процессам посылать сообщения в очередь сообщений, используя для этого всего одну функцию (`msgsnd`), тогда как другие формы IPC требуют вызова трех функций:

open, write и close. Значение идентификатора, присвоенного конкретной очереди, зависит от количества существующих очередей сообщений и от того, сколько раз создавались новые очереди с момента последней перезагрузки ядра. Это динамическое значение – его невозможно предугадать или предопределить в заголовочном файле. Как мы уже говорили в разделе 15.6.1, в простейшем случае сервер должен записать идентификатор очереди в файл, который может быть прочитан клиентами.

Среди других преимуществ очередей сообщений, на которые указывают авторы упоминавшейся выше книги, можно назвать надежность, управление ходом исполнения, ориентированность на отдельные записи и возможность извлекать сообщения не только в порядке их помещения в очередь. Как мы уже видели в разделе 14.4, всеми этими свойствами обладает механизм STREAMS, хотя он и требует вызова функции open перед передачей данных в поток и вызова функции close по окончании работы с потоком. В таблице 15.3 приводятся некоторые сравнительные характеристики различных форм IPC.

Таблица 15.3. Сравнение некоторых характеристик различных форм IPC

Тип IPC	Ориентированность на установление соединения	Надежность	Управление ходом исполнения	Записи	Типы сообщений или свойства
Очереди сообщений	Да	Да	Да	Да	Да
STREAMS	Да	Да	Да	Да	Да
Сокеты домена UNIX, ориентированные на потоки	Да	Да	Да	Нет	Нет
Сокеты домена UNIX, ориентированные на дейтаграммы	Нет	Да	Нет	Да	Нет
FIFO (не STREAMS)	Да	Да	Да	Нет	Нет

(Сокеты, ориентированные на потоки и на дейтаграммы, будут описаны в главе 16. Сокеты домена UNIX будут описаны в разделе 17.3.) Под понятием «ориентированность на установление соединения» подразумевается необходимость предварительного вызова некоторой функции открытия механизма IPC. Как было сказано ранее, мы считаем, что очереди сообщений ориентированы на установление соединения, поскольку для получения идентификатора очереди должны быть предварительно выполнены некоторые действия. Так как область применения всех этих механизмов IPC ограничивается одним хостом, их можно отнести к разряду надежных. Возможность потери сообщений возникает, если сообщения передаются через сеть. Под «управлением ходом исполнения» подразумевается, что передающий процесс может быть приостановлен, если в приемном буфере недостаточно места или принимающий процесс в данное время не может принять сообщение. Когда появится возможность принять сообщение от передающего процесса, его работа будет возобновлена автоматически.

Одна из характеристик, которую мы не упомянули в табл. 15.3, – это возможность автоматически создавать уникальное соединение между сервером и каждым из клиентов. В главе 17 мы увидим, что механизм STREAMS и сокеты, ориентированные на потоки, предоставляют такую возможность.

В следующих трех разделах подробно описываются все три формы XSI IPC.

15.7. Очереди сообщений

Очередь сообщений – это связный список сообщений, который хранится в памяти ядра и идентифицируется идентификатором очереди сообщений. Далее мы будем называть очередь сообщений просто *очередью*, а ее идентификатор – *идентификатором очереди*.

> Стандарт Single UNIX Specification включает определение альтернативной реализации механизма очередей сообщений в расширениях реального времени. Но в этой книге мы не будем рассматривать расширения реального времени.

Создание новой очереди или открытие существующей производится с помощью функции `msgget`. Новые сообщения добавляются в конец очереди функцией `msgsnd`. Каждое сообщение содержит тип (положительное длинное целое число), неотрицательное значение длины и собственно данные (объем которых определяется длиной сообщения). Все эти значения передаются функции `msgsnd` при его добавлении в очередь. Сообщения могут извлекаться из очереди не только в порядке «первым пришел – первым ушел», но и на основе типа сообщения.

С каждой очередью связывается структура `msqid_ds`:

```
struct msqid_ds {
    struct ipc_perm msg_perm; /* раздел 15.6.2 */
    msgqnum_t msg_qnum;       /* количество сообщений в очереди */
    msglen_t msg_qbytes;      /* максимальное количество байт в очереди */
    pid_t msg_lspid;   /* идентификатор процесса, последним вызвавшего msgsnd() */
    pid_t msg_lrpid;   /* идентификатор процесса, последним вызвавшего msgrcv() */
    time_t msg_stime;         /* время последнего вызова msgsnd()*/
    time_t msg_rtime;         /* время последнего вызова msgrcv()*/
    time_t msg_ctime;         /* время последнего изменения */
    .
    .
    .
};
```

Эта структура определяет текущее состояние очереди. Поля структуры, показанные здесь, определяются стандартом Single UNIX Specification. Реализации, как правило, включают в эту структуру дополнительные поля, которые не включаются в стандарт.

В табл. 15.4 перечислены системные пределы, имеющие отношение к очередям сообщений. Если система не поддерживает ту или иную возможность, в соответствующей ячейке указано «не поддерживается». Значение «производное» указывается там, где предел является производным от других преде-

лов. Например, максимальное количество сообщений в Linux зависит от максимального количества очередей и максимального объема данных, которые могут быть помещены в очередь. Если считать, что минимальный размер сообщения составляет 1 байт, тогда максимальное количество сообщений для данной системы может быть вычислено по формуле: *максимальное_количество_очередей* × *максимальный_размер_одной_очереди*. Учитывая значения пределов, которые даны в табл. 15.4, ОС Linux в конфигурации по умолчанию ограничивает количество сообщений числом 262 144. (Даже если сообщение вообще не содержит данных, ОС Linux все равно считает, что такое сообщение имеет размер 1 байт, чтобы ограничить количество сообщений, которые могут быть помещены в очереди.)

Таблица 15.4. Системные пределы, связанные с очередями сообщений

Описание	Типичные значения			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Максимальный размер сообщения	16 384	8 192	Не поддерживается	2 048
Максимальный размер очереди в байтах (т. е. сумма всех сообщений в очереди)	2 048	16 384	Не поддерживается	4 096
Максимальное количество очередей сообщений в системе	40	16	Не поддерживается	50
Максимальное количество сообщений в системе	40	Производное	Не поддерживается	40

> В табл. 15.1 было указано, что Mac OS X 10.3 не поддерживает очереди сообщений XSI. Поскольку Mac OS X частично основана на FreeBSD, а FreeBSD поддерживает очереди сообщений, Mac OS X в принципе также может поддерживать очереди сообщений. С помощью поисковых систем вы наверняка найдете ссылки на реализации очередей сообщений XSI для Mac OS X от сторонних производителей.

Обычно при работе с очередями прежде всего вызывается функция `msgget`, которая открывает существующую или создает новую очередь.

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

int msgget(key_t key, int flag);
```

Возвращает идентификатор очереди в случае успеха, −1 в случае ошибки

В разделе 15.6.1 мы рассмотрели правила преобразования ключа в идентификатор и обсудили вопрос, когда создается новая очередь, а когда открывается существующая. При создании новой очереди инициализируются следующие поля структуры `msqid_ds`:

- Структура `msqid_ds` инициализируется, как описано в разделе 15.6.2. Поле `mode` устанавливается в соответствии со значениями битов прав доступа

в аргументе *flag*. Значения для каждого конкретного права доступа приведены в табл. 15.2.

- В поля msg_qnum, msg_lspid, msg_lrpid, msg_stime и msg_rtime записывается значение 0.
- В поле msg_ctime записывается значение текущего времени.
- В поле msg_qbytes записывается значение соответствующего системного предела.

В случае успеха функция msgget возвращает неотрицательный идентификатор очереди. Это значение может использоваться в других трех функциях, предназначенных для работы с очередями сообщений.

Функция msgctl производит различные операции над очередью. Она и аналогичные ей функции для семафоров и разделяемой памяти (semctl и shmctl) являются аналогами функции ioctl для механизмов XSI IPC.

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

int msgctl(int msqid, int cmd, struct msqid_ds *buf );
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Аргумент *cmd* представляет собой код операции, которая должна быть выполнена над очередью, определяемой аргументом *msqid*.

IPC_STAT Получить структуру msqid_ds данной очереди и сохранить ее по адресу *buf*.

IPC_SET Скопировать из *buf* в структуру msqid_ds, которая связана с очередью, значения следующих полей: msg_perm.uid, msg_perm.gid, msg_perm.mode и msg_qbytes. Эта команда может быть выполнена процессом только в том случае, если его эффективный идентификатор пользователя совпадает со значением msg_perm.cuid или msg_perm.uid или если процесс обладает привилегиями суперпользователя. Значение поля msg_qbytes может увеличить только суперпользователь.

IPC_RMID Удалить очередь сообщений и все данные, которые в ней имеются. Удаление очереди происходит немедленно. Все процессы, которые продолжают использовать очередь, получат код ошибки EIDRM при первой же попытке обращения к ней. Эта команда может быть выполнена процессом только в том случае, если его эффективный идентификатор пользователя совпадает со значением msg_perm.cuid или msg_perm.uid или если процесс обладает привилегиями суперпользователя.

Позже мы увидим, что те же самые команды (IPC_STAT, IPC_SET и IPC_RMID) используются и для управления семафорами и сегментами разделяемой памяти.

Помещение данных в очередь сообщений производится с помощью функции msgsnd.

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

int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Как мы уже упоминали ранее, каждое сообщение состоит из значения, определяющего тип сообщения, поля длины сообщения (*nbytes*) и собственно данных (объем которых равен значению поля длины сообщения).

Через аргумент *ptr* передается указатель на длинное целое со знаком, которое содержит положительное значение, определяющее тип сообщения, за которым сразу же размещаются данные сообщения. (Считается, что сообщение не имеет данных, если в аргументе *nbytes* передается значение 0.) Если максимальный размер отправляемых сообщений составляет 512 байт, то можно определить следующую структуру:

```
struct mymesg {
    long mtype;       /* тип сообщения – положительное число */
    char mtext[512]; /* данные сообщения, объем которых равен nbytes */
};
```

В этом случае в аргументе *ptr* можно передавать указатель на структуру `mymesg`. Тип сообщения может использоваться принимающим процессом для извлечения сообщений в порядке, отличном от порядка помещения сообщений в очередь.

> Некоторые платформы имеют как 32-битную, так и 64-битную реализации. Это сказывается на размере длинных целых чисел и указателей. Например, в 64-битной реализации ОС Solaris допускает сосуществование как 32-битных, так и 64-битных приложений. Если 32-битное приложение попытается выполнить обмен такими структурами данных с 64-битным приложением через неименованный канал или сокет, могут возникнуть проблемы, поскольку размер длинного целого для 32-битных приложений составляет 4 байта, а для 64-битных приложений – 8 байт. Это означает, что 32-битное приложение будет считать, что поле `mtext` отстоит на 4 байта от начала структуры, тогда как 64-битное приложение – что оно отстоит от начала структуры на 8 байт. В этой ситуации часть поля `mtype` 64-битного приложения будет расценена 32-битным приложением как часть поля `mtext`, а первые 4 байта поля `mtext` 32-битного приложения будут расценены 64-битным приложениям как часть поля `mtype`.
>
> Однако эта проблема отсутствует в механизме очередей сообщений XSI. ОС Solaris реализует 32-битные и 64-битные версии системных вызовов IPC с различными точками входа. Системные вызовы заранее предусматривают возможность корректного обмена данными между 32-битными и 64-битными приложениями и правильно интерпретируют размер поля типа сообщения. Единственная проблема, которая здесь может возникнуть, – это потеря информации о типе, когда 64-битное приложение посылает сообщение 32-битному приложению, поскольку 8-байтное поле типа сообщения нельзя без потерь уместить в 4-байтное поле, используемое в 32-битных приложениях. В этом случае 32-битное приложение будет получать усеченное значение типа.

В аргументе *flag* может быть указано значение `IPC_NOWAIT`. Это аналог флага `O_NONBLOCK`, который используется для определения неблокирующего режи-

ма операций файлового ввода-вывода (раздел 14.2). Если очередь сообщений заполнена до отказа (количество сообщений в очереди или общее количество байт в очереди достигло системного предела), при указании флага IPC_NOWAIT функция msgsnd будет сразу же возвращать управление с кодом ошибки EAGAIN. Если флаг IPC_NOWAIT не указан, то процесс будет заблокирован до тех пор, пока не освободится место в очереди, пока очередь не будет удалена из системы или пока не будет перехвачен какой-либо сигнал и обработчик вернет управление. Во втором случае будет возвращен код ошибки EIDRM (identifier removed – идентификатор был удален), а в последнем – код ошибки EINTR.

Обратите внимание на то, как неудачно обрабатывается ситуация удаления очереди. Поскольку для очередей сообщений не поддерживается счетчик ссылок (как для открытых файлов), удаление очереди просто приводит к появлению ошибок при последующих попытках выполнить какие-либо действия с нею. В случае семафоров ситуация удаления обслуживается точно так же. При удалении файла, напротив, его содержимое остается в неприкосновенности до тех пор, пока не будет закрыт последний дескриптор этого файла.

В случае успеха функция msgsnd изменяет содержимое структуры msqid_ds, ассоциированной с заданной очередью: в поле msg_lspid заносится идентификатор вызывающего процесса, в поле msg_stime – время вызова, а значение поля msg_qnum (количество сообщений в очереди) увеличивается на единицу.

Выборка сообщений из очереди производится функцией msgrcv.

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

ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

Возвращает объем данных в сообщении в случае успеха, –1 в случае ошибки

Как и в функции msgsnd, аргумент *ptr* содержит адрес, по которому будет сохранено длинное целое число – тип сообщения, за которым сразу же следует буфер для размещения данных сообщения. Если размер полученного сообщения превышает значение *nbytes* и при этом в аргументе *flag* установлен бит MSG_NOERROR, сообщение будет усечено до размера nbytes. (В этом случае приложение никогда не узнает, что сообщение было усечено.) Если размер полученного сообщения превышает значение *nbytes* и в аргументе *flag* не установлен бит MSG_NOERROR, то вместо сообщения будет возвращен признак ошибки с кодом E2BIG (сообщение при этом останется в очереди).

Аргумент *type* позволяет определить желаемый тип сообщения.

`type == 0` Будет возвращено первое сообщение в очереди.

`type > 0` Будет возвращено первое сообщение, имеющее заданный тип.

`type < 0` Будет возвращено первое сообщение, значение типа которого меньше или равно абсолютному значению аргумента *type*.

Ненулевое значение аргумента *type* используется в том случае, когда необходимо извлекать сообщения из очереди не в порядке их помещения в очередь. Например, значение *type* может указывать приоритет сообщения. Еще один вариант использования поля *type* – клиент может передавать в нем идентификатор своего процесса, если сервер использует единственную очередь для обмена данными со всеми клиентами (разумеется, если идентификатор процесса умещается в длинное целое со знаком).

В аргументе *flag* можно указать значение IPC_NOWAIT, чтобы выполнить операцию в неблокирующем режиме. При наличии этого флага, если в очереди отсутствуют сообщения заданного типа функция msgrcv будет возвращать значение –1 с кодом ошибки ENOMSG в переменной errno. Если флаг IPC_NOWAIT не указан, операция будет заблокирована до тех пор, пока не станет доступно сообщение указанного типа, пока очередь не будет удалена (msgrcv вернет –1 и код ошибки EIDRM в переменной errno) или пока не будет перехвачен сигнал и обработчик сигнала вернет управление (msgrcv вернет –1 и код ошибки EINTR в переменной errno).

В случае успешного завершения функции msgrcv ядро обновит содержимое структуры msqid_ds, ассоциированной с заданной очередью: в поле msg_lrpid будет записан идентификатор вызывающего процесса, в поле msg_rtime – время вызова, а значение поля msg_qnum уменьшится на единицу.

Пример – сравнение производительности очередей сообщений и дуплексных каналов

Для организации двустороннего обмена между клиентом и сервером можно использовать либо очереди сообщений, либо дуплексные каналы. (В табл. 15.1 мы уже упоминали, что дуплексные каналы могут быть организованы на основе сокетов домена UNIX (раздел 17.3), хотя на некоторых платформах имеется поддержка механизма дуплексных каналов на основе функции pipe.)

В табл. 15.5 приводятся результаты сравнения производительности в ОС Solaris трех механизмов: очередей сообщений, каналов STREAMS и сокетов домена UNIX. В процессе измерений тестовая программа создавала канал IPC, вызывала функцию fork, а затем родительский процесс передавал порядка 200 мегабайт данных дочернему процессу. Всего отправлялось 100 000 сообщений по 2 000 байт в каждом. Время приводится в секундах.

Таблица 15.5. Результаты сравнения производительности альтернативных форм IPC в Solaris

Механизм IPC	Пользовательское время	Системное время	Общее время
Очередь сообщений	0,57	3,63	4,22
Канал STREAMS	0,50	3,21	3,71
Сокет домена UNIX	0,43	4,45	5,59

Результаты показывают нам, что очереди сообщений, которые изначально задумывались как скоростной механизм обмена данными, таковым более не являются (фактически, каналы STREAMS имеют более высокую производительность, чем очереди сообщений). (Когда были реализованы очереди сообщений, единственной доступной альтернативной формой IPC были полудуплексные каналы.) При рассмотрении проблем, связанных с очередями сообщений (раздел 15.6.4), мы пришли к выводу, что их не следует использовать в новых приложениях.

15.8. Семафоры

Семафоры не похожи на те формы межпроцессного взаимодействия, которые мы уже описали (именованные и неименованные каналы и очереди сообщений). Семафор – это счетчик, который используется для предоставления доступа к данным, совместно используемым несколькими процессами.

> Стандарт Single UNIX Specification включает определение альтернативного набора функций для работы с семафорами в расширениях реального времени. Но здесь мы не будем обсуждать эти функции.

Чтобы получить доступ к ресурсу, находящемуся в совместном использовании, процесс должен:

1. Проверить состояние семафора, который регулирует доступ к этому ресурсу.
2. Если семафор имеет положительное значение, процесс может обратиться к ресурсу. В этом случае процесс уменьшает значение семафора на 1, указывая тем самым, что он использовал единицу ресурса.
3. В противном случае, если семафор имеет значение 0, процесс приостанавливается до тех пор, пока значение семафора не станет больше 0. После этого процесс возобновит работу и вернется к шагу 1.

По окончании работы с ресурсом, доступ к которому регулируется семафором, значение семафора будет увеличено на 1. Если в этот момент какой-либо другой процесс находится в ожидании освобождения семафора, он возобновит свою работу.

Для корректной работы семафоров необходимо, чтобы проверка состояния семафора и его уменьшение выполнялись в виде атомарной операции. По этой причине семафоры обычно реализуются внутри ядра.

Чаще всего используется разновидность семафоров, которая получила название *двоичный семафор*. Семафоры этого типа регулируют доступ к единственному ресурсу и инициализируются значением 1. Однако вообще семафоры могут инициализироваться любым положительным значением, которое определяет, сколько единиц ресурса, может одновременно использоваться несколькими процессами.

К сожалению, на практике семафоры XSI имеют более сложную организацию. Эта сложность обусловлена следующими тремя особенностями.

1. Семафор – это не просто одиночное неотрицательное значение. Чтобы определить семафор, необходимо определить набор из одного или более семафоров. Количество семафоров в наборе задается при создании этого набора.
2. Создание набора семафоров (`semget`) происходит независимо от его инициализации (`semctl`). Это очень серьезный недостаток, поскольку невозможно атомарно создать новый набор семафоров и инициализировать их значения.
3. Поскольку семафоры, как и все формы XSI IPC, продолжают существовать даже после завершения процессов, их использующих, необходимо предусматривать в программах алгоритмы освобождения размещенных ранее наборов семафоров. В этом может помочь операция *undo*, которая будет описана немного позднее.

Каждому набору семафоров ядро ставит в соответствие структуру `semid_ds`:

```
struct semid_ds {
    struct ipc_perm sem_perm; /* раздел 15.6.2 */
    unsigned short sem_nsems; /* количество семафоров в наборе */
    time_t sem_otime;         /* время последнего вызова функции semop() */
    time_t sem_ctime;         /* время последнего изменения */
    .
    .
    .
};
```

Указанные поля структуры определены стандартом Single UNIX Specification, но реализации могут дополнять структуру `semid_ds` собственными полями.

Каждый из семафоров представлен в наборе анонимной структурой, которая содержит как минимум следующие поля:

```
struct {
    unsigned short semval;  /* значение семафора, всегда >= 0 */
    pid_t sempid;           /* идентификатор процесса, выполнившего */
                            /* последнюю операцию */
    unsigned short semncnt; /* количество процессов, */
                            /* ожидающих выполнения условия semval>curval */
    unsigned short semzcnt; /* количество процессов, */
                            /* ожидающих выполнения условия semval==0 */
    .
    .
    .
};
```

В табл. 15.6 перечислены системные пределы, которые имеют отношение к наборам семафоров.

Обычно при работе с семафорами прежде всего вызывается функция `semget`, которая возвращает идентификатор набора семафоров.

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

int semget(key_t key, int nsems, int flag);
```

Возвращает идентификатор набора семафоров в случае успеха, −1 в случае ошибки

Таблица 15.6. Системные пределы, которые имеют отношение к наборам семафоров

Описание	Типичные значения			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Максимальное значение любого семафора	32 767	32 767	32 767	32 767
Максимальное значение корректировки (adjust-on-exit) для любого семафора (это значение добавляется к семафору при завершении процесса)	16 384	32 767	16 384	16 384
Максимальное количество наборов семафоров в системе	10	128	87 381	10
Максимальное количество семафоров в системе	60	32 000	87 381	60
Максимальное количество емафоров в наборе	60	250	87 381	25
Максимальное количество структур `undo` в системе	30	32 000	87 381	30
Максимальное количество записей в структуре `undo`	10	32	10	10
Максимальное количество операций, выполняемых одним вызовом `semop`	100	32	100	10

В разделе 15.6.1 мы рассмотрели правила преобразования ключа в идентификатор и обсудили вопрос, когда создается новый набор семафоров, а когда открывается существующий. При создании нового набора инициализируются следующие поля структуры `semid_ds`:

- Структура `ipc_perm` инициализируется, как описано в разделе 15.6.2. Поле `mode` устанавливается в соответствии со значениями битов прав доступа в аргументе *flag*. Значения для каждого конкретного права доступа приведены в табл. 15.2.
- В поле `sem_otime` записывается значение 0.
- В поле `sem_ctime` записывается значение текущего времени.
- В поле `sem_nsems` записывается значение аргумента *nsems*.

Количество семафоров в наборе определяется аргументом *nsems*. Если создается новый набор семафоров (обычно на стороне сервера), мы должны указать значение *nsems*. Если открывается существующий набор семафоров, допускается в аргументе *nsems* передавать значение 0.

Операции над набором семафоров выполняются с помощью функции semctl.

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

int semctl(int semid, int semnum, int cmd,
           ... /* union semun arg */);
```

Возвращаемые значения описаны ниже

Четвертый аргумент функции является необязательным и зависит от выполняемой команды; если он присутствует, то представляет собой объединение semun различных аргументов команд:

```
union semun {
    int val;               /* для SETVAL */
    struct semid_ds *buf;  /* для IPC_STAT и IPC_SET */
    unsigned short *array; /* для GETALL и SETALL */
};
```

Обратите внимание, что четвертый аргумент является объединением, а не указателем на объединение.

Аргумент *cmd* определяет одну из следующих десяти операций, которые могут выполняться над набором семафоров, представленным аргументом *semid*. Пять команд, которые используются для работы с отдельным семафором, получают номер семафора в наборе из аргумента *semnum*. Значение *semnum* должно находиться в пределах от 0 до *nsems*–1 включительно.

IPC_STAT Получить структуру semid_ds, которая соответствует заданному набору семафоров, и сохранить ее по адресу *arg.buf*.

IPC_SET Установить значения полей sem_perm.uid, sem_perm.gid и sem_perm.mode в соответствии со значениями этих же полей в структуре, на которую указывает *arg.buf*. Эта команда может быть выполнена процессом только в том случае, если его эффективный идентификатор пользователя совпадает со значением sem_perm.cuid или sem_perm.uid или если процесс обладает привилегиями суперпользователя.

IPC_RMID Удалить набор семафоров. Удаление происходит немедленно. Все процессы, которые продолжают использовать набор семафоров, получат код ошибки EIDRM при первой же попытке обращения к нему. Эта команда может быть выполнена процессом только в том случае, если его эффективный идентификатор пользователя совпадает со значением sem_perm.cuid или sem_perm.uid или если процесс обладает привилегиями суперпользователя.

GETVAL Вернуть значение поля semval для семафора с номером *semnum*.

SETVAL Установить значение поля semval для семафора с номером *semnum*. Значение определяется в *arg.val*.

GETPID Вернуть значение поля sempid для семафора с номером *semnum*.

GETNCNT Вернуть значение поля semncnt для семафора с номером *semnum*.

GETZNCNT Вернуть значение поля semzcnt для семафора с номером *semnum*.

GETALL Вернуть значения всех семафоров в наборе. Значения сохраняются в массиве, на который указывает *arg.array*.

SETALL Установить значения всех семафоров в наборе. Значения берутся из массива, на который указывает *arg.array*.

В случае всех команд GET, за исключением GETALL, функция возвращает соответствующее значение вызывающему процессу. Для остальных команд возвращается 0.

Функция semop выполняет сразу несколько операций над набором семафоров.

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

int semop(int semid, struct sembuf semoparray[], size_t nops);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Аргумент *semoparray* представляет собой массив указателей на операции с семафорами, каждая из которых представлена в виде структуры sembuf:

```
struct sembuf {
    unsigned short sem_num; /* количество семафоров в наборе */
                            /* (0, 1, ..., nsems-1) */
    short sem_op;           /* операция (<0, 0 или >0) */
    short sem_flg;          /* IPC_NOWAIT, SEM_UNDO */
};
```

Аргумент *nops* определяет количество операций (элементов) в массиве.

Операция, выполняемая над каждым семафором из набора, определяется значением sem_op. Это значение может быть отрицательным, положительным или равным нулю. (Ниже мы будем упоминать флаг «undo». Этот флаг соответствует биту SEM_UNDO в поле sem_flg.)

1. Самый простой случай – положительное значение поля sem_op. Он соответствует случаю, когда процесс освобождает занятые ресурсы. Значение sem_op добавляется к значению семафора. Если указан флаг SEM_UNDO, это значение также вычитается из значения корректировки (adjust-on-exit) процесса.
2. Если значение sem_op отрицательное, это означает, что процесс желает получить ресурс, доступ к которому регулируется семафором.

Если значение семафора больше или равно абсолютному значению sem_op (ресурс доступен), абсолютное значение sem_op вычитается из значения семафора. Это гарантирует, что значение семафора ни при каких обстоятельствах не будет меньше нуля. Если указан флаг SEM_UNDO, абсолютное значение sem_op также прибавляется к величине корректировки семафора для данного процесса.

Если значение семафора меньше, чем абсолютное значение sem_op (ресурс недоступен), то вступают в силу следующие условия:

а) Если указан флаг IPC_NOWAIT, функция semop возвращает управление с кодом ошибки EAGAIN.

б) Если флаг IPC_NOWAIT не указан, то для данного семафора увеличивается значение semncnt, а выполнение вызывающего процесса приостанавливается до тех пор, пока не будет соблюдено одно из следующих условий:

- Значение семафора стало больше или равно абсолютному значению sem_op (то есть другой процесс освободил требуемый ресурс). Значение semncnt для этого семафора уменьшается (поскольку ожидание освобождения семафора можно считать законченным), и абсолютное значение sem_op вычитается из значения семафора. Если был указан флаг SEM_UNDO, то абсолютное значение sem_op также добавляется к величине корректировки семафора.
- Семафор был удален из системы. В этом случае функция semop вернет признак ошибки с кодом EIDRM.
- Процессом был перехвачен сигнал, и обработчик сигнала вернул управление. В этом случае значение semncnt уменьшается (поскольку вызывающий процесс больше не ждет освобождения ресурса), и функция semop вернет признак ошибки с кодом EINTR.

3. Если значение sem_op равно нулю, это означает, что процесс желает дождаться момента, когда значение семафора достигнет нуля.

Если значение семафора уже равно нулю, функция сразу же вернет управление.

Если значение семафора больше нуля, тогда вступают в силу следующие условия:

а) Если указан флаг IPC_NOWAIT, функция semop возвращает управление с кодом ошибки EAGAIN.

б) Если флаг IPC_NOWAIT не указан, то для данного семафора увеличивается значение semzcnt и выполнение вызывающего процесса приостанавливается до тех пор, пока не будет соблюдено одно из следующих условий:

- Значение семафора стало равным нулю. В этом случае значение semzcnt уменьшается (поскольку ожидание освобождения семафора можно считать законченным).
- Семафор был удален из системы. В этом случае функция semop вернет признак ошибки с кодом EIDRM.

- Процессом был перехвачен сигнал, и обработчик сигнала вернул управление. В этом случае значение semzcnt уменьшается (поскольку вызывающий процесс прекращает ожидание), и функция semop вернет признак ошибки с кодом EINTR.

Функция semop выполняет все операции атомарно – будут выполнены либо все запрошенные действия, либо ни одно из них.

Корректировка семафора по завершении

Как уже упоминалось ранее, завершение процесса в то время, когда он захватил какие-либо ресурсы посредством семафора, может стать достаточно серьезной проблемой. Всякий раз, когда мы устанавливаем для операции над семафором флаг SEM_UNDO (значение sem_op меньше нуля), ядро запоминает, как много ресурсов было захвачено процессом с помощью конкретного семафора (абсолютное значение sem_op). Когда процесс завершается, добровольно или принудительно, ядро проверяет, имеет ли процесс какие-либо невыполненные корректировки семафоров и, если таковые имеются, корректирует значения соответствующих семафоров.

Когда начальное значение семафора устанавливается функцией semctl с помощью команды SETVAL или SETALL, значение корректировки семафора сбрасывается в 0.

Пример – сравнение производительности семафоров и блокировки записей в файлах

При совместном использовании одного ресурса несколькими процессами порядок доступа к ресурсу может регулироваться с помощью семафора или блокировки записей в файле. Было бы интересно сравнить производительность этих двух методов.

В случае семафоров мы создали набор семафоров, в состав которого входит единственный семафор. Он инициализируется значением 1. Чтобы захватить ресурс, процесс должен вызвать semop со значением sem_op, равным –1. Чтобы освободить ресурс, процесс должен вызвать semop со значением sem_op, равным +1. Кроме того, для каждой операции мы указывали флаг SEM_UNDO на случай завершения процесса, который не успел освободить ресурс.

В случае с блокировками мы создали пустой файл и использовали его первый байт (который не обязательно должен существовать) для установки блокировки. Чтобы захватить ресурс, процесс должен установить блокировку для записи на этот байт, чтобы освободить ресурс – снять блокировку с байта. Одно из свойств блокировок заключается в том, что по завершении процесса, который удерживает блокировку, она будет автоматически снята ядром.

В табл. 15.7 показано время выполнения этих двух методов блокировок в Linux. В каждом случае три тестовых процесса захватывали и освобождали ресурс 100 000 раз. Цифры, приводимые в табл. 15.7, представляют собой общее время для всех трех процессов в секундах.

Таблица 15.7. Производительность двух альтернативных механизмов блокировки в Linux

Механизм IPC	Пользовательское время	Системное время	Общее время
Семафоры с флагом SEM_UNDO	0,38	0,48	0,86
Рекомендательная блокировка записи в файле	0,41	0,95	1,36

В ОС Linux проигрыш при использовании механизма блокировок записей в файлах составил почти 60 процентов по сравнению с семафорами.

Но даже несмотря на то, что механизм блокировок записей медленнее семафоров, в тех случаях, когда речь идет об одном ресурсе (таком как сегмент разделяемой памяти) и вам не нужны все причудливые особенности семафоров XSI, мы все-таки рекомендуем использовать блокировки записей. Причина состоит в том, что они намного проще в использовании и система сама заботится о блокировках, которые не были сняты по завершении процесса.

15.9. Разделяемая память

Механизм разделяемой памяти позволяет двум и более процессам совместно использовать одну и ту же область памяти. Это самый скоростной вид IPC, поскольку при его использовании данные не нужно лишний раз копировать между клиентом и сервером. Единственный сложный момент при работе с разделяемой памятью – это синхронизация доступа к ней. Если сервер размещает некоторые данные в области разделяемой памяти, клиент не должен пытаться читать данные до тех пор, пока сервер не выполнит всю работу. Очень часто для синхронизации используются семафоры. (Но, как мы видели в конце предыдущего раздела, блокировки записей в файлах также могут использоваться.)

> Стандарт Single UNIX Specification включает определение альтернативного набора функций для организации доступа к разделяемой памяти в расширениях реального времени. Но в этой книге мы не будем рассматривать расширения реального времени.

Каждому сегменту разделяемой памяти ядро ставит в соответствие структуру, которая содержит как минимум следующий набор полей:

```
struct shmid_ds {
    struct ipc_perm shm_perm; /* раздел 15.6.2 */
    size_t shm_segsz;  /* размер сегмента в байтах */
    pid_t shm_lpid;    /* идентификатор процесса, последним вызвавшего shmop() */
    pid_t shm_cpid;      /* идентификатор процесса-создателя */
    shmatt_t shm_nattch; /* текущее количество подключений */
    time_t shm_atime;    /* время последнего подключения */
    time_t shm_dtime;    /* время последнего отключения */
    time_t shm_ctime;    /* время последнего изменения */
    .
    .
```

```
};
```

(Каждая реализация при необходимости может добавлять собственные поля в эту структуру.)

Тип `shmatt_t` определен как беззнаковое целое, по меньшей мере — `unsigned short`. В табл. 15.8 перечислены системные пределы (раздел 15.6.3), которые имеют отношение к разделяемой памяти.

Таблица 15.8. Системные пределы, имеющие отношение к разделяемой памяти

Описание	Типичные значения			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Максимальный размер сегмента разделяемой памяти в байтах	33 554 432	33 554 432	4 194 304	8 388 608
Минимальный размер сегмента разделяемой памяти в байтах	1	1	1	1
Максимальное количество сегментов разделяемой памяти в системе	192	4 096	32	100
Максимальное количество сегментов разделяемой памяти для процесса	128	4 096	8	6

Обычно при работе с разделяемой памятью прежде всего вызывается функция `shmget`, которая возвращает идентификатор сегмента разделяемой памяти.

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

int shmget(key_t key, size_t size, int flag);
```

Возвращает идентификатор сегмента разделяемой памяти в случае успеха, −1 в случае ошибки

В разделе 15.6.1 мы рассмотрели правила преобразования ключа в идентификатор и обсудили вопрос, когда создается новый сегмент, а когда открывается существующий. При создании нового сегмента инициализируются следующие поля структуры `shmid_ds`.

- Структура `ipc_perm` инициализируется, как описано в разделе 15.6.2. Поле `mode` устанавливается в соответствии со значениями битов прав доступа в аргументе *flag*. Значения для каждого конкретного права доступа приводятся в табл. 15.2.
- В поля `shm_lpid`, `shm_nattach`, `shm_atime` и `shm_dtime` записывается значение 0.
- В поле `shm_ctime` записывается значение текущего времени.
- В поле `shm_segsz` записывается значение аргумента *size*.

Аргумент *size* определяет размер сегмента разделяемой памяти в байтах. Обычно реализации округляют это число так, чтобы оно было кратно размеру страницы памяти в системе, но если приложение определяет в аргументе size число, не кратное размеру страницы памяти, то остаток последней страницы будет недоступен для использования. Если должен быть создан новый сегмент разделяемой памяти (обычно на стороне сервера), то его размер необходимо определить в аргументе *size*. Если нам нужно лишь получить ссылку на существующий сегмент (в случае клиента), то мы можем передать в аргументе *size* значение 0. Когда создается новый сегмент, его содержимое очищается.

Функция shmctl выполняет различные операции над сегментом разделяемой памяти.

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

int shmctl(int shmid, int cmd, struct shmid_ds *buf );
```

Возвращает 0 в случае успеха, −1 в случае ошибки

Аргумент *cmd* представляет собой код операции, которая должна быть выполнена над сегментом, определяемым аргументом *shmid*.

IPC_STAT Получить структуру shmid_ds для данного сегмента памяти и сохранить ее по адресу *buf*.

IPC_SET Скопировать значения полей shm_perm.uid, shm_perm.gid и shm_perm.mode из *buf* в структуру shmid_ds, связанную с сегментом разделяемой памяти. Эта команда может быть выполнена процессом только в том случае, если его эффективный идентификатор пользователя совпадает со значением shm_ perm.cuid или shm_perm.uid или если процесс обладает привилегиями суперпользователя.

IPC_RMID Удалить сегмент разделяемой памяти. Поскольку для сегментов разделяемой памяти поддерживается счетчик ссылок (поле shm_nattach в структуре shmod_ds), сегмент не будет удален до тех пор, пока последний использующий его процесс не завершится или не отсоединит этот сегмент. Независимо от того, находится ли сегмент в использовании, его идентификатор немедленно удаляется из системы, что предотвращает возможность новых подключений сегмента вызовом функции shmat. Эта команда может быть выполнена процессом только в том случае, если его эффективный идентификатор пользователя совпадает со значением shm_perm.cuid или shm_perm.uid или если процесс обладает привилегиями суперпользователя.

ОС Linux и Solaris предоставляют две дополнительные команды, которые не являются частью стандарта Single UNIX Specification.

SHM_LOCK Заблокировать сегмент разделяемой памяти. Эта команда может быть выполнена, только если процесс обладает привилегиями суперпользователя.

SHM_UNLOCK Разблокировать сегмент разделяемой памяти. Эта команда может быть выполнена, только если процесс обладает привилегиями суперпользователя.

После создания сегмента разделяемой памяти процесс может присоединить его к своему адресному пространству с помощью функции shmat.

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

void *shmat(int shmid, const void *addr, int flag);
```

Возвращает указатель на сегмент разделяемой памяти в случае успеха, –1 в случае ошибки

Адрес, начиная с которого будет присоединен сегмент разделяемой памяти, зависит от значения аргумента *addr* и наличия флага SHM_RND в аргументе *flag*.

- Если в аргументе *addr* передано значение 0, сегмент будет присоединен к первому доступному адресу, который выберет ядро. Это рекомендуемая методика.
- Если в аргументе *addr* передано ненулевое значение и флаг SHM_RND не указан, сегмент присоединяется, начиная с адреса *addr*.
- Если в аргументе *addr* передано ненулевое значение и указан флаг SHM_RND, сегмент будет присоединен с адреса, который вычисляется по формуле: (addr – (addr mod SHMLBA)). Имя константы SHM_RND происходит от слова «round» (округлить), а имя константы SHMLBA, величина которой всегда представлена степенью числа 2, – от «low boundary address multiple» (множитель адреса нижней границы). Приведенная выше формула округляет адрес вниз до ближайшего кратного числу SHMLBA.

Если мы не планируем, что приложение будет работать на единственной аппаратной платформе (что в наши дни весьма маловероятно), мы не должны указывать адрес присоединения сегмента разделяемой памяти. Вместо этого следует передавать в аргументе *addr* значение 0, позволяя системе самой выбрать адрес.

Если в аргументе *flag* указан флаг SHM_RDONLY, присоединенный сегмент будет доступен только для чтения. В противном случае присоединенный сегмент доступен для чтения и записи.

Значение, возвращаемое функцией shmat, представляет собой адрес, начиная с которого был присоединен сегмент разделяемой памяти. В случае ошибки возвращается значение –1. Если вызов shmat завершился успехом, ядро увеличит счетчик shm_nattch в структуре shmid_ds, связанной с данным сегментом.

По окончании работы с сегментом разделяемой памяти следует вызывать функцию shmdt для его отсоединения. Обратите внимание: эта функция не удаляет из системы идентификатор и структуры данных, ассоциированные с сегментом памяти. Идентификатор продолжает существовать до тех пор, пока какой-либо процесс (зачастую сервер) специально не удалит его вызовом функции shmctl с командой IPC_RMID.

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

int shmdt(void *addr);
```

Возвращает 0 в случае успеха, −1 в случае ошибки

В аргументе *addr* передается значение, полученное от функции `shmat`. В случае успеха `shmdt` уменьшает значение счетчика `shm_nattch` в структуре `shmid_ds`.

Пример

Адрес, к которому будет подключен сегмент разделяемой памяти, когда в аргументе `addr` передается значение 0, в значительной степени зависит от операционной системы. Листинг 15.11 содержит текст программы, которая выводит сведения о том, где размещаются различного рода данные в конкретной системе.

Листинг 15.11. Вывод сведений о размещении различного рода данных

```
#include "apue.h"
#include <sys/shm.h>

#define ARRAY_SIZE 40000
#define MALLOC_SIZE 100000
#define SHM_SIZE 100000
#define SHM_MODE 0600                   /* чтение и запись для владельца */

char array[ARRAY_SIZE];                 /* неинициализированные данные = bss */

int
main(void)
{
    int shmid;
    char *ptr, *shmptr;

    printf("array[] от %lx до %lx\n", (unsigned long)&array[0],
           (unsigned long)&array[ARRAY_SIZE]);
    printf("стек примерно %lx\n", (unsigned long)&shmid);

    if ((ptr = malloc(MALLOC_SIZE)) == NULL)
        err_sys("ошибка вызова функции malloc");
    printf("динамически выделенная область от %lx до %lx\n",
           (unsigned long)ptr, (unsigned long)ptr+MALLOC_SIZE);

    if ((shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
        err_sys("ошибка вызова функции shmget");
    if ((shmptr = shmat(shmid, 0, 0)) == (void *)-1)
        err_sys("ошибка вызова функции shmat");
    printf("сегмент разделяемой памяти присоединен в адресах от %lx до %lx\n",
           (unsigned long)shmptr, (unsigned long)shmptr+SHM_SIZE);
    if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("ошибка вызова функции shmctl");
    exit(0);
}
```

Запуск этой программы в ОС Linux на платформе Intel дал следующие результаты:

```
$ ./a.out
array[] от 804a080 до 8053cc0
стек примерно bffff9e4
динамически выделенная область от 8053cc8 до 806c368
сегмент разделяемой памяти присоединен в адресах от 40162000 до 4017a6a0
```

На рис. 15.13 показана раскладка памяти, соответствующая полученным результатам. Обратите внимание: сегмент разделяемой памяти присоединен в адресах, расположенных значительно ниже стека.

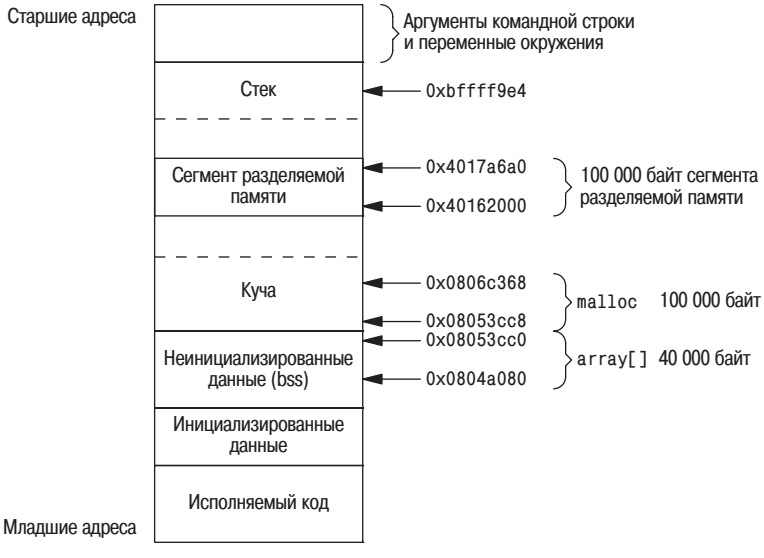


***Рис. 15.13.** Раскладка памяти в ОС Linux на платформе Intel*

В разделе 14.9 мы говорили о том, что с помощью функции `mmap` можно отобразить определенный участок файла в адресное пространство процесса. Концептуально это очень похоже на присоединение сегмента разделяемой памяти с помощью функции `shmat` XSI IPC. Главное отличие состоит в том, что сегмент памяти, отображенный с помощью функции `mmap`, связан с файлом, тогда как сегмент разделяемой памяти XSI вообще никак не связан с файлами.

Пример – отображение в память файла /dev/zero

Разделяемая память может использоваться для организации взаимодействия между процессами, которые не связаны родственными отношениями. Но если процессы взаимосвязаны, то некоторые реализации предоставляют иную методику.

> Следующий прием работает в ОС FreeBSD 5.2.1, Linux 2.4.22 и Solaris 9. В Mac OS X 10.3 в настоящее время отображение символьных устройств в память процесса не поддерживается.

Устройство /dev/zero при чтении из него служит неиссякаемым источником нулевых байтов. Оно также может принимать любые объемы данных, совершенно игнорируя их. Это устройство представляет для нас интерес из-за особых свойств, которые оно проявляет при отображении в память.

- Создается неименованная область памяти, размер которой передается функции mmap во втором аргументе. Это число округляется до ближайшего целого, кратного размеру страницы.
- Область памяти инициализируется нулями.
- Эта область может совместно использоваться несколькими процессами, если их общий предок передал функции mmap флаг MAP_SHARED.

Пример работы с этим устройством приводится в листинге 15.12.

Листинг 15.12. Взаимодействие между родительским и дочерним процессами с использованием операций ввода-вывода над устройством /dev/zero, отображенным в память

```
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

#define NLOOPS 1000
#define SIZE sizeof(long) /* размер сегмента разделяемой памяти */

static int
update(long *ptr)
{
    return((*ptr)++);     /* вернуть значение до увеличения */
}

int
main(void)
{
    int fd, i, counter;
    pid_t pid;
    void *area;

    if ((fd = open("/dev/zero", O_RDWR)) < 0)
        err_sys("ошибка вызова функции open");
    if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
      fd, 0)) == MAP_FAILED)
        err_sys("ошибка вызова функции mmap");
    close(fd); /* теперь, после отображения, /dev/zero можно закрыть */

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid > 0) { /* родительский процесс */
        for (i = 0; i < NLOOPS; i += 2) {
            if ((counter = update((long *)area)) != i)
                err_quit("предок: ожидалось %d, получено %d", i, counter);

            TELL_CHILD(pid);
```

```
            WAIT_CHILD();
        }
    } else { /* дочерний процесс */
        for (i = 1; i < NLOOPS + 1; i += 2) {
            WAIT_PARENT();

            if ((counter = update((long *)area)) != i)
                err_quit("потомок: ожидалось %d, получено %d", i, counter);

            TELL_PARENT(getppid());
        }
    }

    exit(0);
}
```

Эта программа открывает устройство `/dev/zero` и вызывает функцию `mmap`, указывая ей размер отображаемой области. Обратите внимание: когда участок этого специального файла отображен, мы можем закрыть его. После этого создается дочерний процесс. Поскольку при отображении был указан флаг `MAP_SHARED`, данные, которые запишет в эту область один процесс, сможет прочитать другой. (Если бы при отображении мы указали флаг `MAP_PRIVATE`, то этот пример не работал бы.)

Затем родительский и дочерний процессы поочередно начинают увеличивать число, находящееся в разделяемой области отображенной памяти, используя для синхронизации функции из раздела 8.9. Число, находящееся в разделяемой памяти, инициализируется значением 0. Родительский процесс увеличивает его до значения 1, затем дочерний процесс увеличивает его до 2, потом родительский процесс увеличивает его до 3 и т. д. Обратите внимание, что в функции `update` используются круглые скобки, потому что нам нужно увеличить число в памяти, а не сам указатель.

Основное преимущество такого подхода заключается в том, что отпадает необходимость в существовании файла перед созданием отображенной области вызовом `mmap`. Отображение устройства `/dev/zero` автоматически создает область отображенной памяти заданного размера. Недостаток же состоит в том, что такой прием работает только с процессами, которые связаны родственными отношениями. Однако для родственных процессов, вероятно, более простым и эффективным решением было бы использование потоков (главы 11 и 12). Обратите внимание: независимо от выбранной методики, все равно необходимо синхронизировать доступ к разделяемым данным.

Пример – анонимные области отображаемой памяти

Большинство реализаций предоставляют возможность создавать анонимные области отображаемой памяти – примерно так же, как это делается при отображении устройства `/dev/zero`. Чтобы воспользоваться этой возможностью, нужно передать функции `mmap` флаг `MAP_ANON` и число –1 вместо дескриптора файла. В результате мы получим анонимную (поскольку она не связана

с именем какого-либо файла) область памяти, которая может совместно использоваться родственными процессами.

> Возможность создания анонимных областей отображенной памяти имеется во всех четырех платформах, обсуждаемых в этой книге. Обратите внимание на то, что ОС Linux определяет флаг, поддерживающий эту возможность, как MAP_ANONYMOUS, но при этом также определяет и флаг MAP_ANON с тем же самым значением для сохранения совместимости.

Чтобы программа из листинга 15.12 использовала эту возможность, в нее нужно внести три изменения: (а) убрать операцию открытия устройства /dev/zero, (б) убрать операцию закрытия дескриптора и (в) изменить обращение к функции mmap следующим образом:

```
if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
                 MAP_ANON | MAP_SHARED, -1, 0)) == MAP_FAILED)
```

В этом вызове мы указали флаг MAP_ANON и передали значение -1 вместо дескриптора файла. Остальная часть программы из листинга 15.12 остается без изменений.

Последние два примера демонстрируют совместное использование области памяти двумя родственными процессами. Если необходимо использовать разделяемую память для организации взаимодействия между процессами, которые не связаны родственными отношениями, мы можем выбрать один из двух вариантов. Приложения могут использовать функции XSI, предназначенные для работы с разделяемой памятью, или функцию mmap с флагом MAP_SHARED для отображения одного и того же файла в собственные адресные пространства.

15.10. Свойства взаимодействий типа клиент-сервер

Рассмотрим подробнее некоторые свойства клиентов и серверов, которые имеют отношение к различным механизмам IPC, используемым для организации взаимодействия между ними. Самый простой тип взаимоотношений – когда клиент с помощью функций fork и exec запускает требуемый сервер. В этом случае перед вызовом функции fork могут быть созданы два полудуплексных канала, чтобы организовать движение данных в обе стороны. На рис. 15.8 показан пример такой организации взаимодействий. Запускаемый сервер может быть программой с установленным битом set-user-ID, что дает ему специальные привилегии. Кроме того, сервер может идентифицировать клиента, получив свой собственный реальный идентификатор пользователя. (В разделе 8.10 мы говорили, что реальные идентификаторы пользователя и группы не изменяются при запуске новой программы с помощью функции exec.)

На основе этой схемы мы можем разработать *сервер открытия файлов*. (Реализация его будет показана в разделе 17.5.) Он будет открывать файлы по запросу клиента. Таким образом, мы можем добавить проверку дополни-

тельных прав доступа, кроме обычных для UNIX прав пользователь/группа/остальные. Этот подход предполагает, что программа-сервер должна иметь установленный бит set-user-ID, чтобы получить дополнительные привилегии (возможно, привилегии суперпользователя). Сервер на основе реального идентификатора пользователя клиента определяет, разрешен ли ему доступ к запрошенному файлу. Благодаря этому мы можем создать сервер, который предоставляет определенным пользователям дополнительные привилегии, которых они обычно не имеют.

В этом примере, поскольку сервер является дочерним процессом по отношению к клиенту, он может передать родительскому процессу только содержимое файла. Хотя такой подход вполне применим к обычным файлам, он не может быть использован, например, для специальных файлов устройств. Было бы лучше, если бы сервер открывал требуемый файл и передавал клиенту дескриптор этого файла. Родительский процесс может передать дескриптор потомку, но передать дескриптор в обратном направлении, от дочернего процесса родительскому, невозможно (если не использовать специальные приемы, о которых мы расскажем в главе 17).

Следующий тип сервера был показан на рис. 15.12. Этот сервер представляет собой процесс-демон, который взаимодействует со всеми клиентами посредством некоторого механизма IPC. Для такого рода взаимодействий между клиентами и сервером нельзя использовать неименованные каналы. Здесь требуется именованная форма IPC – например, именованные каналы (FIFO) или очереди сообщений. В случае именованных каналов, как мы уже видели, необходимо создавать отдельный именованный канал для связи с каждым из клиентов, если предполагается передача данных клиенту от сервера. Если же данные передаются только от клиента, то достаточно будет создать единственный именованный канал с предопределенным именем. (Такую форму взаимодействия использует демон печати в System V. В этом случае в роли клиента выступает команда `lp`(1), а сервер представлен демоном `lpsched`. Данные в этой схеме передаются только от клиента к серверу, обратная связь полностью отсутствует.)

При использовании очередей сообщений мы получаем дополнительные возможности.

1. Для взаимодействия сервера со всеми клиентами достаточно одной очереди. Поле `type` в сообщении может служить для идентификации получателя сообщения. Например, клиенты могут отправлять запросы, указывая в поле `type` число 1. При этом каждый клиент должен включать в сообщение идентификатор своего процесса. В результате сервер может принимать только сообщения, в которых поле `type` имеет значение 1 (четвертый аргумент функции `msgrcv`), а клиенты – принимать только сообщения, в которых значение поля `type` совпадает с идентификаторами их процессов.
2. Для каждого из клиентов также может быть создана отдельная очередь сообщений. Перед отправкой первого запроса клиент создает собственную очередь сообщений с ключом `IPC_PRIVATE`. Сервер также должен создать очередь с ключом или идентификатором, которые известны клиен-

там. Первый запрос клиент передает через предопределенную очередь сообщений, отсылая серверу идентификатор своей очереди, а весь последующий обмен данными уже будет происходить через отдельную очередь, созданную клиентом. Свой первый и все последующие отклики сервер передает через очередь сообщений клиента.

Один из недостатков такого подхода заключается в том, что каждая клиентская очередь может содержать всего одно сообщение – либо запрос клиента, либо ответ сервера. Это выглядит слишком расточительно из-за ограничений на количество очередей в системе, поэтому вместо отдельных очередей лучше использовать именованные каналы. Другая проблема состоит в том, что сервер вынужден получать сообщения из нескольких очередей сразу, но ни `select`, ни `poll` не могут работать с очередями сообщений.

Любая из этих двух методик, основанных на очередях сообщений, может быть реализована на базе разделяемой памяти с применением методов синхронизации (семафоры или блокировка записей в файле).

Проблема с таким видом взаимодействий клиента и сервера (когда они не связаны родственными отношениями) состоит в том, что сервер должен точно идентифицировать клиента. Если сервер выполняет привилегированные операции, он должен точно знать, кто является клиентом. Это совершенно необходимо, если сервер, например, представляет собой программу с установленным битом set-user-ID. Хотя все эти формы IPC проходят через ядро, оно не предоставляет никаких средств идентификации отправителя.

В случае очередей сообщений, когда для передачи данных между сервером и клиентом используется единственная очередь, в которой может одновременно находиться только одно сообщение, поле `msg_lspid` будет содержать идентификатор процесса отправителя. Но это не совсем то, что нам нужно, желательно было бы иметь эффективный идентификатор пользователя заданного процесса. К сожалению, переносимого способа получения эффективного идентификатора пользователя по идентификатору процесса не существует. (Естественно, ядро хранит оба эти значения в таблице процессов, но, обладая одним, мы не можем получить другой без прямого поиска в памяти ядра.)

В разделе 17.3 мы будем применять следующую методику идентификации клиента на стороне сервера. Этот прием также может использоваться при работе с именованными каналами, очередями сообщений, семафорами или разделяемой памятью. Предположим, что для организации взаимодействий, схема которых представлена на рис. 15.12, используются именованные каналы. Клиент должен создать свой собственный канал FIFO и установить права доступа к нему таким образом, чтобы он был доступен на чтение и на запись только владельцу. Здесь мы исходим из предположения, что сервер обладает привилегиями суперпользователя (в противном случае нет большого смысла беспокоиться по поводу идентификации клиента), таким образом, сервер может выполнять операции чтения и записи с данным каналом. Когда по предопределенному каналу FIFO поступает первый запрос от клиента (который должен содержать идентификатор канала клиента), сервер вызывает функцию `stat` или `fstat` для канала клиента. Предполагается, что эф-

фективный идентификатор пользователя клиента – это идентификатор владельца FIFO (поле st_uid структуры stat). Сервер должен убедиться в том, что доступ к каналу разрешен только для его владельца. Дополнительно сервер должен проверить, имеют ли три поля времени, связанные с FIFO (поля st_atime, st_mtime и st_ctime структуры stat), допустимые значения (например, не более 15 или 30 секунд). Если злоумышленник сможет создать канал FIFO с другим эффективным идентификатором и установить право только на чтение и на запись для владельца, значит, система имеет весьма серьезные проблемы с безопасностью.

Чтобы реализовать эту методику для XSI IPC, вспомните, что с каждой очередью сообщений, семафором и сегментом разделяемой памяти ассоциируется структура ipc_perm, которая идентифицирует создателя объекта IPC (поля cuid и cgid). Как и в случае FIFO, сервер должен требовать от клиента, чтобы создаваемая им структура IPC имела права доступа только для владельца. Кроме того, сервер должен убедиться в том, что все характеристики времени имеют надлежащие значения (поскольку эти структуры IPC могут существовать в системе до тех пор, пока явно не будут удалены).

В разделе 17.2.2 мы увидим, что существует более надежный способ идентификации, когда эффективные идентификаторы пользователя и группы клиента предоставляются ядром. Сделать это можно с помощью подсистемы STREAMS, передавая дескрипторы файлов между процессами.

15.11. Подведение итогов

Мы рассмотрели разнообразные формы взаимодействий между процессами: именованные и неименованные каналы и три формы IPC, которые обычно называют XSI IPC (очереди сообщений, семафоры и разделяемую память). Семафоры в действительности представляют собой механизм синхронизации, а не обмена данными, и часто используются для синхронизации доступа к разделяемым ресурсам, таким как сегменты разделяемой памяти. При обсуждении неименованных каналов мы рассмотрели реализацию функции popen, понятие сопроцессов и возможные ловушки, связанные с режимом буферизации в стандартной библиотеке ввода-вывода.

После сравнения производительности очередей сообщений с дуплексными каналами и семафоров с механизмом блокировки записей в файлах мы можем дать следующие рекомендации. Изучайте именованные и неименованные каналы, поскольку эти два механизма по-прежнему остаются эффективным средством организации обмена данными для большинства приложений. Избегайте использования очередей сообщений и семафоров в новых приложениях. Вместо них следует применять дуплексные каналы и блокировки записей в файлах, так как они намного проще. Разделяемая память может найти применение, хотя те же самые возможности предоставляются функцией mmap (раздел 14.9).

В следующей главе мы рассмотрим механизмы сетевых взаимодействий, которые помогают организовать обмен информацией между разными машинами.

Упражнения

15.1. В программе из листинга 15.2 удалите вызов функции close перед вызовом waitpid в конце кода родителя. Объясните, что произойдет.

15.2. В программе из листинга 15.2 удалите обращение к функции waitpid в конце кода родителя. Объясните, что произойдет.

15.3. Что произойдет, если функции popen передать имя несуществующей команды? Напишите небольшую программу, чтобы проверить эту ситуацию.

15.4. В программе из листинга 15.9 удалите обработчик сигнала, запустите программу и завершите дочерний процесс. Каким образом можно убедиться, что родительский процесс завершился при получении сигнала SIGPIPE после ввода строки?

15.5. Попробуйте в программе из листинга 15.9 использовать для работы с неименованными каналами вместо функций read и write функции чтения и записи из стандартной библиотеки ввода-вывода.

15.6. В пояснениях к стандарту POSIX.1 в качестве одной из причин появления функции waitpid приводится описание ситуации, которая не может быть обработана без этой функции:

```
if ((fp = popen("/bin/true", "r")) == NULL)
    ...
if ((rc = system("sleep 100")) == -1)
    ...
if (pclose(fp) == -1)
    ...
```

Что получится в результате выполнения этого кода, если вместо функции waitpid использовать функцию wait?

15.7. Объясните, как функции select и poll обрабатывают ситуацию закрытия неименованного канала пишущим процессом. Чтобы ответить на этот вопрос, напишите две небольшие программы: одну с использованием функции select, другую с использованием функции poll.

15.8. Что произойдет, если команда *cmdstring*, запущенная функцией popen со значением "r" в аргументе *type*, попытается вывести что-нибудь на стандартный вывод сообщений об ошибках?

15.9. Для выполнения команды из аргумента *cmdstring* функция popen вызывает командный интерпретатор. Что происходит по завершении *cmdstring*? (Подсказка: нарисуйте схему происходящего.)

15.10. Стандарт POSIX.1 особо отмечает, что возможность открытия канала FIFO с помощью функции open одновременно для чтения и записи не предусмотрена, хотя большинство версий UNIX допускают это. Продемонстрируйте другой метод открытия FIFO для чтения и записи без использования блокировок.

15.11. Если файл не содержит секретной информации, то его доступность на чтение для всех пользователей не несет никакого вреда. (Хотя обычно попытки совать нос в чужие файлы не одобряются.) Но что может произойти, если злонамеренный процесс получит доступ на чтение к очереди сообщений, которая используется для взаимодействия сервера и нескольких клиентов? Какой информацией должен обладать злонамеренный процесс, чтобы прочитать содержимое очереди сообщений?

15.12. Напишите программу, которая выполняет следующие действия: пять раз в цикле создает очередь сообщений, выводит идентификатор очереди, удаляет очередь сообщений; затем в другом цикле пять раз создает очередь сообщений с ключом `IPC_PRIVATE` и размещает в очереди одно сообщение. После завершения программы просмотрите очереди сообщений с помощью команды `ipcs`(1). Объясните, что происходит с идентификаторами очередей.

15.13. Опишите, как можно создать связанный список объектов данных в сегменте разделяемой памяти. Что следует хранить в качестве указателей в списке?

15.14. Нарисуйте временную диаграмму работы программы из листинга 15.12, показывающую значение переменной `i` в родительском и дочернем процессах, значения числа в разделяемой памяти и возвращаемые значения функции `update`. Исходите из предположения, что после вызова функции `fork` первым получает управление дочерний процесс.

15.15. Перепишите программу из листинга 15.12 таким образом, чтобы она вместо отображаемой памяти использовала функции для работы с разделяемой памятью XSI из раздела 15.9.

15.16. Перепишите программу из листинга 15.12 таким образом, чтобы она использовала семафоры для синхронизации родительского и дочернего процессов.

15.17. Перепишите программу из листинга 15.12 таким образом, чтобы она использовала механизм блокировки записей в файле для синхронизации родительского и дочернего процессов.