Министерство науки и высшего образования Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

---

А.В. ГУНЬКО

# СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ В СРЕДЕ LINUX

Утверждено Редакционно-издательским советом университета
в качестве учебного пособия

НОВОСИБИРСК
2020

УДК 004.45 (075.8)
Г 948

Рецензенты:

*А.А. Малявко*, канд. техн. наук, доцент
*А.Б. Колкер*, канд. техн. наук, доцент

**Гунько А.В.**

Г 948 Системное программирование в среде Linux: учебное пособие / А.В. Гунько. – Новосибирск: Изд-во НГТУ, 2020. – 235 с.

ISBN 978-5-7782-4160-2

В данном пособии описан интерфейс прикладного программирования (API) UNIX-совместимых операционных систем: от файловых операций и использования библиотек до методов и средств разработки многозадачного и многопоточного программного обеспечения, а также средства межзадачной (IPC) и межпоточной коммуникации: программные каналы и каналы FIFO, очереди сообщений, семафоры, разделяемая память System V и POSIX, взаимные исключения и условные переменные.

Кроме того, кратко обсуждаются средства коммуникации процессов по сети и особенности взаимодействия приложений и системных служб.

Учебное пособие предназначено для студентов IV курса, обучающихся по направлению 27.03.04 «Управление в технических системах», а также может быть полезно студентам ряда других технических специальностей, связанных с разработкой многозадачного и многопоточного программного обеспечения в среде операционных систем семейства Linux.

Работа подготовлена на кафедре автоматики НГТУ

**УДК 004.45 (075.8)**

**ISBN 978-5-7782-4160-2**

# Предисловие

Данное учебное пособие можно рассматривать как вторую часть учебного пособия «Программирование (в среде Windows)» [1], поскольку в нем с тех же позиций описывается примерно тот же набор системных вызовов, характерных для любой современной операционной системы (ОС).

Пособие базируется на существенно переработанном конспекте лекций по дисциплине «Системное программное обеспечение» [2], читаемой автором с 2004 г. Переработка заключалась не только в удалении глав, относящихся к Windows API, но и в добавлении разделов, посвященных низкоуровневым операциям с файлами, программированию и применению статических и динамических библиотек функций, средств IPC стандарта POSIX, сетевому программированию, системным службам (демонам).

Издание ориентировано на студентов, умеющих программировать на классическом (ANSI C) языке C, имеющих представление об общих принципах построения современных операционных систем и знающих основные команды UNIX-систем.

# Глава 1. Введение в системное программирование

Любая современная операционная система (ОС) всегда выступает как интерфейс между аппаратной частью компьютера и пользователем с его прикладными программами, которым предоставляется среда для их выполнения. Под интерфейсом операционных систем следует понимать специальные интерфейсы системного и прикладного программирования, предназначенные для выполнения следующих задач [3]:

1. Управление процессами, которое включает в себя следующий набор основных функций:

- запуск, приостановка и снятие задачи с выполнения;
- задание или изменение приоритета задачи;
- взаимодействие задач между собой (используя механизмы сигналов, семафоры, очереди, каналы);
- RPC (remote procedure call) – удаленный вызов подпрограмм.

2. Управление памятью:

- запрос на выделение блока памяти;
- освобождение памяти;
- изменение параметров блока памяти (память может быть заблокирована процессом либо предоставлена в общий доступ);
- отображение файлов на память (имеется не во всех системах).

3. Управление вводом-выводом:

- запрос на управление виртуальными устройствами (управление вводом-выводом является привилегированной функцией ОС);
- файловые операции (запросы к системе управления файлами на создание, изменение и удаление данных, собранных в файлы).

Обычно этот интерфейс называют ядром (kernel), так как оно имеет относительно небольшой объем и составляет основу ОС. На рис. 1.1 изображена схема, отражающая архитектуру системы UNIX [4].

Интерфейс ядра – это слой программного обеспечения, называемый системными вызовами (область с заливкой на рис. 1.1). Библиотеки функций общего пользования основываются на интерфейсе системных вызовов, но прикладная программа может свободно пользоваться как теми, так и другими. Командная оболочка (shell) – это особое приложение, которое предоставляет интерфейс для запуска других приложений.

В более широком смысле операционная система – это ядро и все остальное программное обеспечение, которое делает компьютер пригодным к использованию и обеспечивает его индивидуальность. В состав этого программного обеспечения входят системные утилиты, прикладные программы, командные оболочки, библиотеки функций общего пользования и т. п.


*Рис. 1.1*. Архитектура системы UNIX

Любая операционная система дает прикладным программам возможность обращаться к системным службам. Во всех реализациях UNIX имеется строго определенное число точек входа в ядро, которые называются системными вызовами (см. рис. 1.1). В Linux с ядром версии 3.2.0 имеется 380 системных вызовов, а в FreeBSD 8.0 их более 450.

Интерфейс системных вызовов определяется на языке С независимо от конкретных реализаций, использующих системные вызовы в той или иной системе. В этом его отличие от многих старых систем, которые традиционно определяли точки входа в ядро на языке ассемблера.

В системе UNIX для каждого системного вызова предусматривается одноименная функция в стандартной библиотеке языка С (библиотечная функция). Пользовательский процесс вызывает эту функцию как обычно, а она вызывает соответствующую службу ядра, применяя способ обращения, принятый в данной системе. Далее будем рассматривать системные вызовы и библиотечные функции как обычные функции языка С. И те и другие предназначены для обслуживания прикладных программ. Однако при этом нужно понимать, что библиотечные функции можно заменить, если в этом возникнет необходимость, а системные вызовы – нет. Соответственно, под ***системным программированием*** будем понимать программирование прикладных приложений с использованием системных вызовов.

Рассмотрим в качестве примера функцию выделения памяти **`malloc`**. Существует масса способов распределения памяти и алгоритмов «сборки мусора», но нет единой методики, оптимальной абсолютно для всех возможных ситуаций. Системный вызов **`sbrk`** (см. **`man 2 sbrk`** или **`info sbrk`** в вашей UNIX-системе), описанный в файле включения unistd.h, который занимается выделением памяти, не является диспетчером памяти общего назначения. Он лишь увеличивает или уменьшает адресное пространство процесса на заданное количество байтов, а управление этим пространством возлагается на сам процесс. Функция **`malloc`** реализует одну конкретную модель распределения памяти. На самом деле многие программные пакеты реализуют собственные алгоритмы распределения памяти с использованием системного вызова **`sbrk`**. На рис. 1.2 показаны взаимоотношения между приложением, функцией **`malloc`** и системным вызовом **`sbrk`**.

Здесь мы видим четкое разделение обязанностей: системный вызов выделяет дополнительную область памяти от имени процесса, а библиотечная функция **`malloc`** распоряжается этой областью.

Прикладная программа может обращаться к системному вызову и к библиотечной функции. Кроме того, следует помнить, что библиотечные функции в свою очередь также могут обращаться к системным вызовам. Это наглядно показано на рис. 1.3.

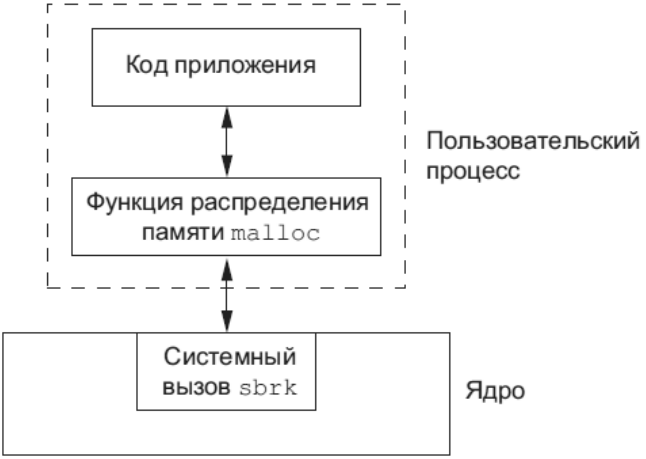



*Рис. 1.2*. Разделение обязанностей функции **`malloc`** и системного вызова **`sbrk`**

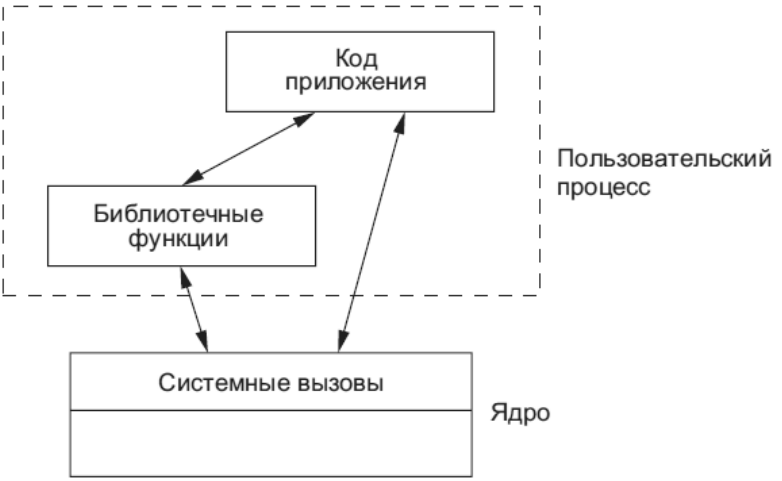



*Рис. 1.3.* Разделение обязанностей библиотечных функций и системных вызовов

Другое отличие системных вызовов от библиотечных функций заключается в том, что системные вызовы обеспечивают лишь минимально необходимую функциональность, тогда как библиотечные функции часто обладают более широкими возможностями. Мы уже видели это различие на примере сравнения системного вызова **sbrk** с библиотечной функцией **malloc**. Мы еще столкнемся с этим различием, когда будем сравнивать функции низкоуровневого ввода-вывода и стандартные функции ввода-вывода (глава 3).

Системные вызовы управления процессами (**fork**, **exec** и **waitpid**, см. главу 5) обычно вызываются пользовательским процессом напрямую. Но существуют также библиотечные функции, которые служат для упрощения самых распространенных случаев: например, функция **popen**, которая будет рассмотрена в разделе 6.2.

## Вопросы для самопроверки

1. Перечислите основные задачи, решаемые программными интерфейсами операционных систем.
2. Что понимается под термином «ядро ОС»?
3. Дайте иное, чем описано выше, определение системных вызовов.
4. Что понимается под термином «системное программирование»?
5. В чем разница между библиотечными функциями и системными вызовами?

# Глава 2. Общие принципы Linux API

## 2.1. Стандарты, лежащие в основе Linux API

Первая реализация UNIX была разработана в 1969 году (в год рождения Линуса Торвальдса (Linus Torvalds)) Кеном Томпсоном (Ken Thompson) в компании Bell Laboratories, являвшейся подразделением телефонной корпорации AT&T. Эта реализация была написана на ассемблере для мини-компьютера Digital PDP-7. К 1973 году язык C уже был доведен до состояния, позволившего почти полностью переписать на нем ядро UNIX. Таким образом, UNIX стала одной из самых ранних ОС, написанных на языке высокого уровня, что позволило в дальнейшем портировать ее на другие аппаратные архитектуры.

Версия UNIX, включавшая в себя свой собственный исходный код, получила весьма широкое распространение под названием Berkeley Software Distribution (BSD). Первым полноценным дистрибутивом, появившимся в декабре 1979 года, стал 3BSD.

В 1981 году состоялся выпуск UNIX System III (три). Эта версия была создана организованной в компании AT&T группой поддержки UNIX (UNIX Support Group, USG).

В 1983 году последовал первый выпуск System V (пять). Несколько последующих выпусков привели к тому, что в 1989 году состоялся окончательный выпуск System V Release 4 (SVR4), ко времени которого в System V было перенесено множество свойств из BSD, включая сетевые объекты. Лицензия на System V была выдана множеству коммерческих поставщиков, использовавших эту версию как основу своих собственных реализаций UNIX.

Ввиду широкого разнообразия реализаций UNIX, основанных как на BSD, так и на System V, усложнилось портирование программных продуктов с одной реализации UNIX на другую. Эта си-

туация показала, что требовалась стандартизация языка программирования С и системы UNIX, чтобы упростить портирование приложений с одной системы на другую. Рассмотрим выработанные в итоге стандарты.

### 2.1.1. Стандарт языка С

Стандарт языка С, на котором написано ядро UINX, приводился в вышедшей в 1978 году книге Кернигана (Kernighan) и Ритчи (Ritchie) «Язык программирования Си». С появлением в 1985 году языка С++ проявились конкретные улучшения и дополнения, которые могли быть привнесены в С без нарушения совместимости с существующими программами. В частности, сюда можно отнести прототипы функций, присваивание структур, спецификаторы типов (const и volatile), перечисляемые типы и ключевое слово void. Эти факторы побудили к стандартизации языка С. Ее кульминацией в 1989 году стало утверждение Американским институтом национальных стандартов (ANSI) стандарта языка С (X3.159-1989), который в 1990 году был принят в качестве стандарта (ISO/IEC 9899:1990) Международной организацией по стандартизации (ISO). Наряду с определением синтаксиса и семантики языка С в этом стандарте давалось описание стандартной библиотеки С, включающей возможности stdio, функции обработки строк, математические функции, различные файлы заголовков и т. д. Эту версию С обычно называют С89 или (значительно реже) ISO С90, и она полностью рассмотрена во втором издании (1988 года) книги Кернигана и Ритчи «Язык программирования Си».

Пересмотренное издание стандарта языка С было принято ISO в 1999 году. Его обычно называют С99, и он включает несколько изменений языка и его стандартной библиотеки. В частности, там описаны добавление типов данных long long и логического (булева), присущий С++ стиль комментариев (//), ограниченные указатели и массивы переменной длины.

Исторически С89 часто называли ANSI С, и это название до сих пор иногда употребляется в таком значении. Но после того, как комитет ANSI принял пересмотренную версию С99, будет правильным считать, что стандартом ANSI С следует называть С99.

Стандарты языка С не зависят от особенностей операционной системы, то есть они не привязаны к UNIX-системе. Это означает, что программы на языке С, для написания которых использовалась только

стандартная библиотека С, должны быть портированы на любой компьютер и операционную систему, предоставляющую реализацию языка С.

### 2.1.2. Стандарты POSIX

Термин POSIX (аббревиатура от Portable Operating System Interface) обозначает группу стандартов, разработанных под руководством Института инженеров электротехники и электроники – Institute of Electrical and Electronic Engineers (IEEE), а точнее, его комитета по стандартам для портируемых приложений – Portable Application Standards Committee (PASC, http://www.pasc.org). Цель PASC-стандартов – содействие портируемости приложений на уровне исходного кода. Последняя буква X появилась потому, что названия большинства вариантов UNIX заканчивались на X.

POSIX.1 стал IEEE-стандартом в 1988 году и после небольшого количества пересмотров был принят как стандарт ISO в 1990 году (ISO/IEC 9945-1:1990). В POSIX.1 документируется интерфейс прикладного программирования (API) для набора сервисов, которые должны быть доступны программам на языке С, но при этом не требуется, чтобы с этим интерфейсом была связана какая-либо конкретная реализация. Это означает, что интерфейс может быть реализован любой операционной системой, и не обязательно UNIX. ОС, способная справиться с этой задачей, может быть сертифицирована в качестве совместимой с POSIX.1.

Стандарт IEEE POSIX 1003.1c (POSIX.1c), одобренный в 1995 году, содержит определения потоков в POSIX. Стандарт IEEE POSIX 1003.1g (POSIX. lg) определил API для работы в сети, включая сокеты. Стандарты IEEE POSIX 1003.Id (POSIX.Id), одобренный в 1999 году, и POSIX.lj, одобренный в 2000 году, определили дополнительные расширения основного стандарта POSIX для работы в режиме реального времени.

В декабре 2001 года был одобрен стандарт POSIX 1003.1-2001, иногда называемый просто POSIX.1-2001. Этот стандарт также известен как Single UNIX Specification версии 3, и ссылки на него будут в основном иметь вид SUSv3. Базовая спецификация SUSv3 состоит почти из 3700 страниц, разбитых на следующие четыре части.

1. Base Definitions (XBD). Включает в себя определения, термины, положения и спецификации содержимого файлов заголовков. Всего предоставляются спецификации 84 файлов заголовков.

2. System Interfaces (XSH). Начинается с различной полезной справочной информации. В основном в ней содержатся спецификации разных функций (реализуемых либо в виде системных вызовов, либо в виде библиотечных функций в конкретной реализации UNIX). Всего в нее включено 1123 системных интерфейса.

3. Shell and Utilities (XCU). В этой части определяются возможности оболочки и различные команды (утилиты) UNIX. Всего в ней представлено 160 утилит.

4. Rationale (XRAT). Включает в себя текстовые сведения и объяснения, касающиеся предыдущих частей.

Кроме того, в SUSv3 входит спецификация X/Open CURSES Issue 4 версии 2 (XCURSES), в которой определяются 372 функции и три файла заголовков для API curses, предназначенного для управления экраном.

В 2008 году был завершен масштабный пересмотр стандарта, что привело к объединенному стандарту POSIX 1003.1-2008 и SUSv4. Основную часть стандарта занимает описание системных вызовов.

### 2.1.3. Стандарты LSB

В 1985 году был основан Фонд свободного программного обеспечения – Free Software Foundation (FSF), некоммерческая организация для поддержки проекта GNU (рекурсивно определяемый акроним, взятый из фразы GNU's not UNIX), а также для разработки совершенно свободного ПО. Когда был запущен проект GNU, в его понятиях версия BSD не была свободной. Для использования BSD по-прежнему требовалось получить лицензию от AT&T, и пользователи не могли свободно изменять и распространять дальше код AT&T, формирующий часть BSD.

Одним из важных результатов появления проекта GNU была разработка общедоступной лицензии – GNU General Public License (GPL). Большинство программных средств в дистрибутиве Linux, включая ядро, распространяются под лицензией GPL. Программное обеспечение, распространяемое под лицензией GPL, должно быть доступно в форме исходного кода и должно предоставлять право дальнейшего распространения в соответствии с положениями GPL. Внесение изменений в программы, распространяемые под лицензией, не запрещено, но любое распространение такой измененной программы должно также производиться в соответствии с положениями о GPL-лицензи-

ровании. Если измененное программное средство распространяется в исполняемом виде, автор также должен дать возможность приобрести измененные исходные коды. Первая версия GPL была выпущена в 1989 году.

Значительная часть программного кода, составляющего то, что обычно называют системой Linux, фактически была взята из проекта GNU. Говоря «Linux», обычно подразумевают полноценную UNIX-подобную операционную систему, часть которой формируется ядром Linux. Если называть вещи своими именами, то название Linux относится лишь к ядру, разработанному Линусом Торвальдсом. И тем не менее сам термин Linux обычно используется для обозначения ядра, а также широкого ассортимента других программных средств (инструментов и библиотек), которые в совокупности составляют полноценную операционную систему.

На самых ранних этапах существования Linux пользователю требовалось собрать все эти инструменты воедино, создать файловую систему и правильно разместить и настроить в ней все программные средства. В результате появился рынок для распространителей Linux. Они проектировали пакеты (дистрибутивы) для автоматизации основной части процесса установки, создания файловой системы и установки ядра, а также других требуемых системе программных средств.

Самые первые дистрибутивы появились в 1992 году. Самый старый из выживших до сих пор коммерческих дистрибутивов, Slackware, появился в 1993 году. Примерно в то же время появился и некоммерческий дистрибутив Debian, за которым вскоре последовали SUSE и Red Hat. В настоящее время весьма большой популярностью пользуется дистрибутив Ubuntu, который впервые появился в 2004 году. Разработчики Linux (то есть ядра, библиотеки glibc и инструментария) стремятся соответствовать различным стандартам UNIX, особенно POSIX и Single UNIX Specification (SUS).

Из-за наличия нескольких распространителей Linux, а также из-за того, что реализаторы ядра не контролируют содержимое дистрибутивов, стандартной коммерческой версии Linux не существует. Ядро отдельного дистрибутива Linux обычно базируется на некоторой версии ядра Linux из основной ветки разработки с набором необходимых изменений.

Итогом стали в основном незначительные различия в системах, предлагаемых разными компаниями – распространителями Linux.

В результате усилий по обеспечению совместимости между разными дистрибутивами Linux появился стандарт под названием Linux Standard Base (LSB). В рамках LSB был разработан и внедрен набор стандартов для систем Linux. Они обеспечивают возможность запуска двоичных приложений (то есть скомпилированных программ) на любой LSB-совместимой системе. Двоичная портируемостъ, внедренная с помощью LSB, отличается от портируемости исходного кода, внедренной стандартом POSIX. Портируемость исходного кода означает возможность написания программы на языке C с ее последующей успешной компиляцией и запуском на любой POSIX-совместимой системе. Двоичная совместимость имеет куда более привередливый характер, и, как правило, она недостижима на различных аппаратных платформах. Она позволяет осуществлять однократную компиляцию на конкретной аппаратной платформе, после чего запускать откомпилированную программу в любой совместимой реализации, запущенной на этой аппаратной платформе. Двоичная портируемость является весьма важным требованием для коммерческой жизнеспособности приложений, созданных под Linux независимыми поставщиками программных продуктов – independent software vendor (ISV).

Подводя итоги, можно сказать, что Linux API базируется на стандартах ANSI C (C99), POSIX 1003.1-2008 (SUSv4) и LSB.

## 2.2. Задачи, выполняемые Linux API

Понятие «*операционная система*» зачастую употребляется в двух различных значениях.

1. Для обозначения всего пакета, содержащего основные программные средства управления ресурсами компьютера и все вспомогательные стандартные программные инструменты: интерпретаторы командной строки, графические пользовательские интерфейсы, файловые утилиты и редакторы.

2. В более узком смысле – для обозначения основных программных средств, управляющих ресурсами компьютера (например, центральным процессором, оперативной памятью и устройствами) и занимающихся их распределением.

В качестве синонима второго значения зачастую используется понятие «*ядро*».

### 2.2.1. Задачи, выполняемые ядром

Кроме всего прочего, в круг задач, выполняемых ядром, входят следующие.

*Диспетчеризация процессов*. У компьютера имеется один или несколько центральных процессоров (CPU), выполняющих инструкции программ. Как и другие UNIX-системы, Linux является многозадачной операционной системой с вытеснением. Многозадачность означает, что несколько процессов (например, запущенные программы) могут одновременно находиться в памяти и каждая может получить в свое распоряжение центральный процессор (процессоры). Правила, определяющие, какие именно процессы получают в свое распоряжение центральный процессор (ЦП) и на какой срок, устанавливает имеющийся в ядре диспетчер процессов (а не сами процессы).

*Управление памятью*. Как и в большинстве современных операционных систем, в Linux используется управление *виртуальной памятью* – технология, дающая два основных преимущества.

1. Процессы изолированы друг от друга и от ядра, поэтому один процесс не может читать или изменять содержимое памяти другого процесса или ядра.

2. В памяти требуется хранить только часть процесса, снижая таким образом объем памяти, требуемый каждому процессу, и позволяя одновременно содержать в оперативной памяти большее количество процессов. Вследствие этого повышается эффективность использования центрального процессора, так как в результате увеличивается вероятность того, что в любой момент времени есть по крайней мере один процесс, который может быть выполнен центральным процессором (процессорами).

*Предоставление файловой системы*. Ядро предоставляет файловую систему на диске, позволяя создавать, считывать обновлять, удалять файлы, выполнять их выборку и производить с ними другие действия.

*Создание и завершение процессов* (*управление задачами*). Ядро может загрузить новую программу в память, предоставить ей ресурсы (например, центральный процессор, память и доступ к файлам), необходимые для работы. Такой экземпляр запущенной программы называется *процессом* (*задачей*). Как только выполнение процесса завершится, ядро обеспечивает высвобождение используемых им ресурсов для дальнейшего применения другими программами.

*Доступ к устройствам* (*управление вводом-выводом*). Устройства (мыши, мониторы, клавиатуры, дисковые и ленточные накопители и т. д.), подключенные к компьютеру, позволяют обмениваться информацией между компьютером и внешним миром – осуществлять ввод-вывод данных. Ядро предоставляет программы с интерфейсом, упрощающим доступ к устройствам. Этот доступ происходит в рамках определенного стандарта. Одновременно с этим ядро распределяет доступ к каждому устройству со стороны нескольких процессов.

*Работа в сети.* Ядро от имени пользовательских процессов отправляет и принимает сетевые сообщения (пакеты). Эта задача включает в себя маршрутизацию сетевых пакетов в направлении целевой операционной системы.

*Предоставление интерфейса прикладного программирования* (*API*) *системных вызовов.* Процессы могут запрашивать у ядра выполнение различных задач с использованием точек входа в ядро, известных как *системные вызовы*. API системных вызовов Linux – главная тема данной книги. Этапы выполнения процессом системного вызова подробно описаны далее.

Кроме перечисленных выше свойств такая многопользовательская операционная система, как Linux, обычно предоставляет пользователям абстракцию *виртуального персонального компьютера*. Иначе говоря, каждый пользователь может зайти в систему и работать в ней практически независимо от других. Например, у каждого пользователя имеется собственное дисковое пространство (домашний каталог). Кроме этого, пользователи могут запускать программы, каждая из которых получает свою долю времени центрального процессора и работает со своим виртуальным адресным пространством. Эти программы в свою очередь могут независимо друг от друга получать доступ к устройствам и передавать информацию по сети. Ядро занимается разрешением потенциальных конфликтов при доступе к ресурсам оборудования, поэтому пользователи и процессы обычно даже ничего о них не знают.

### 2.2.2. Режим ядра и пользовательский режим

Современные вычислительные архитектуры обычно позволяют центральному процессору работать как минимум в двух различных режимах: *пользовательском* и *режиме ядра* (который иногда называют *защищенным*). Аппаратные инструкции позволяют переключаться из

одного режима в другой. Соответственно области виртуальной памяти могут быть помечены в качестве части *пользовательского пространства* или *пространства ядра.* При работе в пользовательском режиме ЦП может получать доступ только к той памяти, которая помечена в качестве памяти пользовательского пространства. Попытки обращения к памяти в пространстве ядра приводят к выдаче аппаратного исключения. При работе в режиме ядра центральный процессор может получать доступ как к пользовательскому пространству памяти, так и к пространству ядра.

*Отличия пользовательских процессов и процессов ядра.* В работающей системе обычно выполняется множество процессов. Для процесса многое происходит асинхронно. Выполняемый процесс не знает, когда он будет приостановлен в следующий раз, для каких других процессов будет спланированно время центрального процессора (и в каком порядке) или когда в следующий раз это время будет спланировано для него. Передача сигналов и возникновение событий обмена данными между процессами осуществляется через ядро и может произойти в любое время. Многое происходит незаметно для процесса. Он не знает, где находится в памяти, размещается ли конкретная часть его пространства памяти в самой оперативной памяти или же в области подкачки (в выделенной области дискового пространства, используемой для дополнения оперативной памяти компьютера). Точно так же процесс не знает, где на дисковом накопителе хранятся файлы, к которым он обращается, – он просто ссылается на файлы по имени. Процесс работает изолированно, он не может напрямую обмениваться данными с другим процессом. Он не может сам создать новый процесс или даже завершить свое собственное существование. И наконец, процесс не может напрямую обмениваться данными с устройствами ввода-вывода, подключенными к компьютеру.

С другой стороны, у работающей системы имеется всего одно ядро, которое обо всем знает и всем управляет. Ядро содействует выполнению всех процессов в системе. Оно решает, какой из процессов следующим получит доступ к центральному процессору, когда это произойдет и сколько продлится. Ядро обслуживает структуры данных, содержащие информацию обо всех запущенных процессах, и обновляет их по мере создания процессов, изменения их состояния и прекращения их выполнения. Ядро обслуживает все низкоуровневые структуры данных, позволяющие преобразовывать имена файлов, используемые программами, в физические местоположения файлов на диске. Ядро

также обслуживает структуры данных, которые отображают виртуальную память каждого процесса в физическую память компьютера и в область (области) подкачки на диске. Весь обмен данными между процессами осуществляется через механизмы, предоставляемые ядром. Отвечая на запросы процессов, ядро создает новые процессы и прекращает работу существующих. И наконец, ядро (в частности, драйверы устройств) выполняет весь непосредственный обмен данными с устройствами ввода-вывода, осуществляя по требованию перемещение информации в пользовательские процессы и из них в устройства.

Далее, встречая фразы вроде «процесс может создавать другой процесс», «процесс может создать канал», «процесс может записывать данные в файл», следует помнить, что посредником во всех этих действиях является ядро, а такие утверждения – лишь сокращения фраз типа «процесс может *запросить у ядра* создание другого процесса» и т. д.

Говоря простым языком, *процесс* представляет собой экземпляр выполняемой программы. Когда программа выполняется, ядро загружает ее код в виртуальную память, выделяет память под переменные программы и определяет учетные структуры данных ядра для записи различной информации о процессе (имеются в виду идентификатор процесса, код завершения, пользовательские и групповые идентификаторы).

С точки зрения ядра процессы являются объектами, между которыми ядро должно делить различные ресурсы компьютера. В случае с ограниченными ресурсами, например памятью, ядро изначально выделяет некоторый их объем процессу и регулирует это выделение в ходе жизненного цикла процесса, реагируя на потребности процесса и общие потребности системы в этом ресурсе. Когда процесс завершается, все такие ресурсы высвобождаются для повторного использования другими процессами. Другие ресурсы, такие как время центрального процессора и сетевой трафик, являются возобновляемыми, но должны быть поровну поделены между всеми процессами.

### 2.2.3. Модель памяти процесса, его создание и выполнение

Процесс логически делится на следующие части, известные как *сегменты*:

- *текст* – инструкции программы;

- *данные* – статические переменные, используемые программой;
- *динамическая память* (*куча*) – область, из которой программа может динамически выделять дополнительную память;
- *стек* – часть памяти, которая может расширяться и сжиматься по мере вызова функций и возвращения из них и которая используется для выделения хранилища под локальные переменные и информацию о взаимосвязанности вызовов функций.

Процесс может создать новый процесс с помощью системного вызова **`fork()`**. Процесс, вызывающий **`fork()`**, известен *как родительский процесс,* а новый процесс называется *дочерним процессом.* Ядро создает дочерний процесс путем изготовления дубликата родительского процесса. Дочерний процесс наследует копии родительских сегментов данных, стека и кучи, которые затем могут изменяться независимо от своих родительских копий. Текст программы размещается в области памяти с пометкой «только для чтения» и совместно используется двумя процессами.

Дочерний процесс запускается либо для выполнения другого набора функций в том же самом коде, что и у родительского процесса, либо зачастую для использования системного вызова **`execve()`** с целью загрузки и выполнения совершенно новой программы. Вызов **`execve()`** удаляет существующие сегменты текста, данных, стека и кучи, заменяя их новыми сегментами, основываясь на коде новой программы.

У вызова **`execve()`** есть ряд надстроек в виде родственных функций библиотеки языка С с несколько отличающимся интерфейсом, но со сходной функциональностью. У всех этих функций имена начинаются со строки **`exec`**.

В основном глагол «выполнять» (**`exec`**) будет употребляться для описания операций, выполняемых **`execve()`** и ее библиотечными функциями-надстройками.

### 2.2.4. Особенности выполнения системных вызовов Linux API

*Системный вызов* представляет собой управляемую точку входа в ядро, позволяющую процессу запрашивать у ядра осуществление некоторых действий в интересах процесса. Ядро дает возможность программам получать доступ к некоторым сервисам с помощью интер-

фейса прикладного программирования (API) системных вызовов. К таким сервисам, к примеру, относятся создание нового процесса, выполнение ввода-вывода и создание каналов для межпроцессного взаимодействия.

Перед тем как перейти к подробностям работы системных вызовов, следует упомянуть о некоторых их общих характеристиках.

- Системный вызов изменяет состояние процессора, переводя его из пользовательского режима в режим ядра, позволяя таким образом центральному процессору получать доступ к защищенной памяти ядра.
- Набор системных вызовов не изменяется. Каждый системный вызов идентифицируется по уникальному номеру. (Обычно программам эта система нумерации неизвестна, они идентифицируют системные вызовы по именам.)
- У каждого системного вызова может быть набор аргументов, определяющих информацию, которая должна быть передана из пользовательского пространства (то есть из виртуального адресного пространства процесса) в пространство ядра и наоборот.

С точки зрения программирования инициирование системного вызова во многом похоже на вызов функции языка С. Но при выполнении системного вызова многое происходит закулисно. Процесс выполнения системного вызова подробно описан в [5].

В Linux подпрограммы обслуживания системных вызовов следуют соглашению о том, что для указания успеха возвращается неотрицательное значение. При ошибке подпрограмма возвращает отрицательное число, являющееся значением одной из **`errno`**-констант с противоположным знаком. Когда возвращается отрицательное значение, функция-оболочка библиотеки С меняет его знак на противоположный (делая его положительным), копирует результат в **`errno`** и возвращает значение –1, чтобы указать вызывающей программе на возникновение ошибки.

## 2.3. Системные типы данных Linux API

При использовании стандартных типов языка С для конкретной реализации ОС предоставляются различные типы данных, например идентификаторы процессов. Конечно, для объявления переменных, хранящих подобную информацию, можно было бы использовать основные типы языка С, например **`int`** и **`long`**, но это сокращает воз-

можность портирования между системами UNIX по следующим причинам:

- Размеры этих основных типов от реализации к реализации UNIX отличаются друг от друга (например, **long** в одной системе может занимать 4 байта, а в другой – 8 байт). Кроме того, в разных реализациях для представления одной и той же информации могут использоваться различные типы. Например, в одной системе идентификатор процесса может быть типа **int**, а в другой – типа **long**.
- Даже в одной и той же реализации UNIX типы, используемые для представления информации, могут в разных выпусках отличаться друг от друга. Наглядными примерами в Linux могут послужить идентификаторы пользователей и групп. В Linux 2.2 и более ранних версиях эти значения были представлены в 16 разрядах. В Linux 2.4 более поздних версиях они представлены в виде 32-разрядных значений.

В SUSv3 указываются различные стандартные типы системных данных, а к реализации предъявляются требования по надлежащему определению и использованию этих типов. Каждый из этих типов определен с помощью имеющегося в языке C спецификатора **typedef**. Например, тип данных **pid_t** предназначен для представления идентификаторов процессов и в Linux/x86-32 определяется следующим образом:

```
typedef int pid_t;
```

У большинства стандартных типов системных данных имена оканчиваются на _t. Многие из них объявлены в заголовочном файле <sys/types.h>, хотя некоторые объявлены в других заголовочных файлах. Приложение должно использовать эти определения типов, чтобы портируемым образом объявить используемые им переменные. Например, следующее объявление позволит приложению правильно представить идентификаторы процессов в любой совместимой с SUSv3 системе:

```
pid_t mypid;
```

В таблице перечислены типы системных данных, которые будут встречаться далее. Для отдельных типов в этой таблице SUSv3 требует, чтобы они были реализованы в качестве арифметических типов. Это означает, что при реализации в качестве базового типа может быть выбран либо целочисленный тип, либо тип с плавающей точкой (вещественный или комплексный).

**Отдельные типы системных данных**

| Тип данных | Требование к типу в SUSv3 | Описание |
|---|---|---|
| id_t | Целое число | Базовый тип для хранения идентификаторов; достаточно большой, по крайней мере для pid_t, uid_t и gid_t |
| in_addr_t | 32-разрядное целое число без знака | IPv4-адрес |
| in_port_t | 16-разрядное целое число без знака | Номер порта IP |
| ino_t | Целое число без знака | Номер индексного дескриптора файла |
| key_t | Арифметический тип | Ключ IPC в System V |
| mode_t | Целое число | Тип файла и полномочия доступа к нему |
| mqd_t | Требования к типу отсутствуют, но не должен быть типом массива | Дескриптор очереди сообщений POSIX |
| msglen_t | Целое число без знака | Количество байтов, разрешенное в очереди сообщений в System V |
| msgqnum_t | Целое число без знака | Количество сообщений в очереди сообщений в System V |
| off_t | Целое число со знаком | Смещение в файле или размер файла |
| pid_t | Целое число со знаком | Идентификатор процесса, группы процессов или сессии |
| ptrdiff_t | Целое число со знаком | Разница между двумя значениями указателей в виде целого числа со знаком |
| rlim_t | Целое число без знака | Ограничение ресурса |

Окончание таблицы

| Тип данных | Требование к типу в SUSv3 | Описание |
|---|---|---|
| sa_family_t | Целое число без знака | Семейство адресов сокета |
| shmatt_t | Целое число без знака | Количество прикрепленных процессов для совместно используемого сегмента памяти System V |
| size_t | Целое число без знака | Размер объекта в байтах |
| socklen_t | Целочисленный тип, состоящий как минимум из 32 разрядов | Размер адресной структуры сокета в байтах |
| ssize_t | Целое число со знаком | Количество байтов или (при отрицательном значении) признак ошибки |
| suseconds_t | Целое число со знаком в разрешенном диапазоне | Интервал времени в микросекундах |
| time_t | Целое число или вещественное число с плавающей точкой | Календарное время в секундах от начала отсчета времени |
| timer_t | Арифметический тип | Идентификатор таймера для функций временных интервалов POSIX.lb |
| uid_t | Целое число | Числовой идентификатор пользователя |

При выводе значений одного из типов системных данных, показанных в таблице (например, **pid_t** и **uid_t**), нужно проследить, чтобы в вызов функции **printf()** не была включена зависимость представления данных. Она может возникнуть из-за того, что имеющиеся в языке С правила расширения аргументов приводят к преобразованию значений типа **short** в **int**, но оставляют значения типа **int** и **long** в неизменном виде. Иными словами, в зависимости от определения типа системных данных вызову **printf()** передается либо **int**, либо **long**. Но поскольку функция **printf()** не может определять типы в

ходе выполнения программы, вызывающий код должен предоставить эту информацию в явном виде, используя спецификатор формата **%d** или **%ld**. Проблема в том, что простое включение в программу одного из этих спецификаторов внутри вызова **printf()** создает зависимость от реализации. Обычно применяется подход, при котором используется спецификатор **%ld**, с неизменным приведением соответствующего значения к типу **long**:

```
pid_ t mypid;
mypid = getpid (); / * Возвращает идентификатор процесса * /
printf("My PID is % ld \n " , (long ) mypid);
```

Для указанного выше подхода следует сделать одно исключение. Поскольку в некоторых средах компиляции тип данных **off_t** имеет размерность **long long**, мы приводим **off_t** -значения к этому типу и используем спецификатор **%lld**.

## Вопросы для самопроверки

1. Перечислите стандарты, лежащие в основе Linux API.
2. Каковы основные задачи, решаемые ядром Linux?
3. В чем заключаются отличия пользовательского процесса и процесса ядра?
4. Каковы общие характеристики системных вызовов?
5. Каковы этапы выполнения системных вызовов?
6. Каковы особенности вывода на экран значений системных типов данных?

# Глава 3. Файловые операции средствами системных вызовов

Перейдем к подробному рассмотрению API системных вызовов. Лучше всего начать с файлов, поскольку они лежат в основе всей философии UNIX. Основное внимание здесь будет уделено системным вызовам, предназначенным для выполнения файлового ввода-вывода.

Вначале будет описан дескриптор файла, а затем рассмотрены системные вызовы, составляющие так называемую универсальную модель ввода-вывода. Это те самые системные вызовы, которые открывают и закрывают файл, а также считывают и записывают данные. Большая часть этой информации важна для усвоения последующего материала, поскольку те же самые системные вызовы используются для выполнения ввода-вывода во всех типах файлов, в том числе каналах.

## 3.1. Общее представление о файловом вводе-выводе

Все системные вызовы для выполнения ввода-вывода совершаются в отношении открытых файлов с использованием *дескриптора файла,* представленного неотрицательным (обычно небольшим) целым числом. Дескрипторы файлов применяются для обращения ко всем типам открытых файлов, включая программные каналы, каналы FIFO, сокеты, терминалы, аппаратные устройства и обычные файлы. Каждый процесс имеет свой собственный набор дескрипторов файлов.

Обычно от большинства программ ожидается возможность использования трех стандартных дескрипторов файлов, перечисленных в табл. 3.1. Эти три дескриптора открыты оболочкой от имени программы еще до запуска самой программы. В интерактивной оболочке эти три дескриптора обычно ссылаются на терминал, на котором запущена

оболочка. Если в командной строке указано перенаправление ввода-вывода, то оболочка перед запуском программы обеспечивает соответствующее изменение дескрипторов файлов.

Таблица 3.1

**Стандартные дескрипторы файлов**

| Дескриптор файла | Назначение | Имя согласно POSIX | Поток stdio |
|---|---|---|---|
| 0 | Стандартный ввод | **STDIN_FILENO** | **stdin** |
| 1 | Стандартный вывод | **STDOUT_FILENO** | **stdout** |
| 2 | Стандартная ошибка | **STDERR_FILENO** | **stderr** |

При ссылке на эти дескрипторы файлов в программе можно использовать либо номера (0, 1 или 2), либо, что предпочтительнее, стандартные имена POSIX, определенные в файле <unistd.h>.

Рассмотрим следующие четыре системных вызова, которые являются ключевыми для выполнения файлового ввода-вывода (языки программирования обычно используют их исключительно опосредованно, через библиотеки ввода-вывода).

1. **fd = open (pathname, flags, mode)** – открытие файла, идентифицированного по имени пути к нему – **pathname**, с возвращением дескриптора файла, который используется для обращения к открытому файлу в последующих вызовах. Если файл не существует, вызов **open()** может его создать в зависимости от установки битовой маски аргумента флагов – **flags**. В аргументе флагов также указывается, с какой целью открывается файл: для чтения, для записи или для проведения обеих операций. Аргумент **mode** (режим), определяет права доступа, которые будут накладываться на файл, если он создается этим вызовом. Если вызов **open()** не будет использоваться для создания файла, этот аргумент игнорируется и может быть опущен.

2. **numread = read (fd, buffer, count)** – считывание не более указанного в **count** количества байтов из открытого файла, ссылка на который дана в **fd**, и сохранение их в буфере **buffer**. При вызове **read()** возвращается количество фактически считанных байтов. Если данные не могут быть считаны (то есть встретился конец файла), **read()** возвращает 0.

3. **numwritten = write (fd, buffer, count)** – запись из буфера байтов, количество которых указано в **count**, в открытый файл, ссылка на который дана в **fd**. При вызове **write()** возвращается количество фактически записанных байтов, которое может быть меньше значения, указанного в **count**.

4. **status = close (fd)** – вызывается после завершения ввода-вывода с целью высвобождения дескриптора файла **fd** и связанных с ним ресурсов ядра.

Перед тем как подробно разбирать эти системные вызовы, посмотрим на небольшую демонстрацию их использования в листинге 3.1. Эта программа является простой версией команды **cp (man 1 cp)**. Она доступна для скачивания на сайте [6], посвященном данной дисциплине, в разделе «Примеры программ», «Файлы». Программа копирует содержимое существующего файла, чье имя указано в первом аргументе командной строки, в новый файл с именем, указанным во втором аргументе командной строки.

Листинг 3.1. Использование системных вызовов ввода-вывода

```
// copy.c
#include <sys/stat.h>
#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>
#define BUF_SIZE 1024
int main (int argc, char * argv [ ])
{
int inputFd, outputFd, openFlags;
mode_t filePerms ;
ssize_t numRead;
char buf[BUF_SIZE];
if (argc != 3)
{
printf("Usage: %s old-file new-file \n", argv[0]);
exit(-1);
}
/* Открытие файлов ввода и вывода */
inputFd = open (argv[1], O_RDONLY);
if (inputFd == -1)
{
```

```
printf ("Error opening file %s\n", argv[1]); exit(-2);
}
openFlags = O_CREAT | O_WRONLY | O_TRUNC;
filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
S_IROTH | S_IWOTH; /* rw - rw - rw - */
outputFd = open (argv [2], openFlags, filePerms);
if (outputFd == -1)
{
printf ("Error opening file %s\n ", argv[2]);
exit(-3);
}
/* Перемещение данных до достижения конца файла
ввода или возникновения ошибки */
while ((numRead = read (inputFd, buf, BUF_SIZE)) > 0)
{
if (write (outputFd, buf, numRead) != numRead)
{
printf ("couldn't write whole buffer\n "); exit(-4);
}
if (numRead == -1)
{
printf ("read error\n "); exit(-5);
}
if (close (inputFd ) == -1 )
{
printf ("close input error\n"); exit(-6);
}
if (close (outputFd ) == -1 )
{
printf ("close output error\n"); exit(-7);
}
}
exit(EXIT_SUCCESS);
}
```

## 3.2. Универсальность ввода-вывода

Одна из отличительных особенностей модели ввода-вывода UNIX *состоит в универсальности ввода-вывода.* Это означает, что одни и те же четыре системных вызова – **open()**, **read()**, **write()** и

**close()** – применяются для выполнения ввода-вывода во всех типах файлов, включая устройства, например терминалы. Следовательно, если программа написана с использованием лишь этих системных вызовов, она будет работать с любым типом файла. Например, следующие примеры показывают вполне допустимое использование программы, чей код приведен в листинге 3.1:

```
$./copy a.txt /dev/tty - Копирование обычного файла в этот терминал
$./copy /dev/tty b.txt - Копирование ввода с этого терминала в обычный файл
$./copy /dev/pts/16 /dev/tty - Копирование ввода с другого терминала
```

Универсальность ввода-вывода достигается обеспечением того, что в каждой файловой системе и в каждом драйвере устройства реализуется один и тот же набор системных вызовов ввода-вывода. Поскольку детали реализации конкретной файловой системы или устройства обрабатываются внутри ядра, при написании прикладных программ мы можем вообще игнорировать факторы, относящиеся к устройству. Когда требуется получить доступ к конкретным свойствам файловой системы или устройства, в программе можно использовать всеобъемлющий системный вызов **ioctl()**. Он предоставляет интерфейс для доступа к свойствам, которые выходят за пределы универсальной модели ввода-вывода.

## 3.3. Открытие файла: open ( )

Системный вызов **open()** либо открывает существующий файл, либо создает и открывает новый файл. Возвращает дескриптор при успешном завершении или –1 при ошибке:

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

Чтобы файл открылся, он должен быть найден по аргументу **pathname**. Если в этом аргументе находится символьная ссылка, она разыменовывается. В случае успеха **open()** возвращает дескриптор файла, который используется для ссылки на файл в последующих системных вызовах. В случае ошибки **open()** возвращает –1.

Аргумент **flags** является битовой маской, указывающей *режим доступа* к файлу с использованием одной из констант, перечисленных в табл. 3.2.

Когда **open()** применяется для создания нового файла, аргумент битовой маски режима (**mode**) указывает на права доступа, которые должны быть присвоены файлу. (Используемый тип данных **mode_t** является целочисленным типом, определенным в SUSv3.) Если при вызове **open()** не указывается флаг **O_CREAT**, то аргумент **mode** может быть опущен.

Таблица 3.2

**Режимы доступа к файлам**

| Режим доступа | Описание |
|---|---|
| **O_RDONLY** | Открытие файла только для чтения |
| **O_WRONLY** | Открытие файла только для записи |
| **O_RDWR** | Открытие файла как для чтения, так и для записи |

Аргумент **mode** может быть указан в виде числа (обычно восьмеричного) или, что более предпочтительно, путем применения операции логического ИЛИ ( | ) к нескольким константам битовой маски, перечисленным в табл. 3.3.

Таблица 3.3

**Константы для битов прав доступа к файлу**

| Константа | Восьмеричное значение | Бит прав доступа |
|---|---|---|
| **S_IRUSR** | 0400 | Пользователь: чтение |
| **S_IWUSR** | 0200 | Пользователь: запись |
| **S_IXUSR** | 0100 | Пользователь: выполнение |
| **S_IRGRP** | 040 | Группа: чтение |
| **S_IWGRP** | 020 | Группа: запись |
| **S_IXGRP** | 010 | Группа: выполнение |
| **S_IROTH** | 04 | Остальные: чтение |
| **S_IWOTH** | 02 | Остальные: запись |
| **S_IXOTH** | 01 | Остальные: выполнение |

В листинге 3.2 показаны примеры использования **open()**. В некоторых из них указываются дополнительные биты флагов, которые будут рассмотрены далее.

Листинг 3.2. Примеры использования **open()**

```
/* Открытие существующего файла для чтения */
fd = open ("startup", O_RDONLY);
/* Открытие нового или существующего файла
для чтения и записи с усечением до нуля байтов;
предоставление владельцу исключительных прав
доступа на чтение и запись * /
fd = open ("myfile", O_RDWR | O_CREAT | O_TRUNC,
S_IRUSR | S_IWUSR);
/* Открытие нового или существующего файла
для записи; записываемые данные должны всегда
добавляться в конец файла * /
fd = open ("w.log" , O_WRONLY | O_CREAT | O_APPEND,
S_IRUSR | S_IWUSR);
```

В некоторых примерах вызова **open()**, показанных в листинге 3.2, во флаги кроме режима доступа к файлу включены дополнительные биты (**O_CREAT**, **O_TRUNC** и **O_APPEND**). Рассмотрим аргумент **flags** более подробно. В табл. 3.4 приведен основной набор констант, любая комбинация которых с помощью побитового ИЛИ ( | ) может быть передана в аргументе **flags**. В последнем столбце показано, какие из этих констант были включены в стандарт SUSv3 или SUSv4.

Таблица 3.4

**Значения для аргументов флагов системного вызова open()**

| Флаг | Назначение | SUS |
|---|---|---|
| **O_RDONLY** | Открытие только для чтения | v3 |
| **O_WRONLY** | Открытие только для записи | v3 |
| **O_RDWR** | Открытие для чтения и записи | v3 |
| **O_CREAT** | Создание файла, если он еще не существует | v3 |
| **O_DIRECTORY** | Отказ, если аргумент pathname указывает не на каталог | v4 |

Окончание табл. 3.4

| Флаг | Назначение | SUS |
|---|---|---|
| **O_EXCL \| O_CREAT** | Исключительное создание файла | v3 |
| **O_NOFOLLOW** | Запрет на разыменование символьных ссылок | v4 |
| **O_TRUNC** | Усечение существующего файла до нулевой длины | v3 |
| **O_APPEND** | Записи добавляются исключительно в конец файла | v3 |
| **O_DIRECT** | Операции ввода-вывода осуществляются без использования кеша | v3 |
| **O_NONBLOCK** | Открытие в неблокируемом режиме | v3 |

В случае возникновения ошибки при попытке открытия файла системный вызов **open()** возвращает –1, а в **errno** идентифицируется причина ошибки. Далее перечислены основные возможные ошибки, которые могут произойти:

- **EACCES** – права доступа к файлу не позволяют вызывающему процессу открыть файл в режиме, указанном флагами. Из-за прав доступа к каталогу доступ к файлу невозможен или файл не существует и не может быть создан;
- **EISDIR** – указанный файл является каталогом, а вызывающий процесс пытается открыть его для записи. Это запрещено;
- **EMFILE** – достигнуто ограничение ресурса процесса на количество файловых дескрипторов (**RLIMIT_NOFILE**);
- **ENFILE** – достигнуто ограничение на количество открытых файлов, накладываемое на всю систему;
- **ENOENT** – заданный файл не существует и ключ **O_CREAT** не указан; или **O_CREAT** был указан и один из каталогов в путевом имени не существует или является символьной ссылкой, ведущей на несуществующее путевое имя (битой ссылкой);
- **EROFS** – указанный файл находится в файловой системе, предназначенной только для чтения, а вызывающий процесс пытается открыть его для записи;

- **ETXTBSY** – заданный файл является исполняемым (программой) и в данный момент выполняется. Изменение исполняемого файла, связанного с выполняемой программой (то есть его открытие для записи), запрещено. (Чтобы изменить исполняемый файл, сначала следует завершить программы.)

При дальнейшем описании других системных вызовов или библиотечных функций мы не будем перечислять возможные ошибки, которые могут произойти при подобных обстоятельствах. Этот перечень можно найти на соответствующих страницах руководства для каждого системного вызова или библиотечной функции, да и приведенный выше список неполон: дополнительные причины отказа **open()** можно найти на странице руководства **open** (**man 2 open**).

## 3.4. Чтение из файла: read( )

Системный вызов **read()** позволяет считывать данные из открытого файла, на который ссылается дескриптор **fd**:

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

Функция возвращает количество считанных байтов, 0 при **EOF** или –1 при ошибке.

Аргумент **count** определяет максимальное количество считываемых байтов (тип данных **size_t** – беззнаковый целочисленный). Аргумент **buffer** предоставляет адрес буфера памяти, в который должны быть помещены входные данные. Этот буфер должен иметь длину в байтах не менее той, что задана в аргументе **count**.

Системные вызовы не выделяют память под буферы, которые используются для возвращения информации вызывающему процессу. Вместо этого следует передать указатель на ранее выделенный буфер памяти подходящего размера. Этим вызовы отличаются от ряда библиотечных функций, которые выделяют буферы в памяти с целью возвращения информации вызывающему процессу.

При успешном вызове **read()** возвращается количество фактически считанных байтов или 0, если встретился символ конца файла. При ошибке обычно возвращается –1. Тип данных **ssize_t** относится к целочисленному типу со знаком. Этот тип используется для хранения количества байтов или значения –1, которое служит признаком ошибки.

При вызове **read()** количество считанных байтов может быть меньше запрашиваемого. Возможная причина для обычных файлов – близость считываемой области к концу файла. При использовании вызова **read()** в отношении других типов файлов, например программных каналов, каналов FIFO, сокетов или терминалов, также могут складываться различные обстоятельства, при которых количество считанных байтов оказывается меньше запрашиваемого. Например, изначально применение **read()** в отношении терминала приводит к считыванию символов только до следующего встреченного символа новой строки ( **\n** ).

```
#define MAX_READ 20
char buffer[MAX_READ];
if (read(STDIN_FILENO, buffer, MAX_READ) == -1)
exit (-1) ;
printf ("The input data was: %s \n", buffer);
```

Этот фрагмент кода выведет весьма странные данные, поскольку в них, скорее всего, будут включены символы, дополняющие фактически введенную строку. Дело в том, что вызов **read()** не добавляет завершающий нулевой байт в конце строки, которая задается для вывода функции **printf()**. Нетрудно догадаться, что именно так и должно быть, поскольку **read()** может использоваться для чтения любой последовательности байтов из файла. В некоторых случаях входные данные могут быть текстом, но бывает, что это двоичные целые числа или структуры языка С в двоичном виде. Невозможно «объяснить» вызову **read()** разницу между ними, поэтому он не в состоянии выполнять соглашение языка С о завершении строки символов нулевым байтом. Если в конце буфера входных данных требуется наличие завершающего нулевого байта, его нужно вставлять явным образом:

```
char buffer[MAX_READ + 1];
ssize_t numRead;
numRead = read(STDIN_FILENO, buffer, MAX_READ);
if (numRead == -1 ) exit (-1) ;
buffer[numRead] = ' \ 0 ‘ ;
printf ("The input data was: %s \n", buffer);
```

Поскольку для завершающего нулевого байта требуется байт памяти, размер буфера должен быть как минимум на один байт больше максимальной предполагаемой считываемой строки.

## 3.5. Запись в файл: write( )

Системный вызов **write()** записывает данные в открытый файл и возвращает количество записанных байтов или –1 при ошибке:

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

Аргументы для **write()** аналогичны тем, что использовались для **read()**: **buffer** представляет собой адрес записываемых данных, **count** является количеством записываемых из буфера данных, а **fd** содержит дескриптор файла, который ссылается на тот файл, куда будут записываться данные.

В случае успеха вызов **write()** возвращает количество фактически записанных данных, которое может быть меньше значения аргумента **count**. Для дискового файла возможными причинами такой частичной записи может оказаться переполнение диска или достижение ограничения ресурса процесса на размеры файла. (Речь идет об ограничении **RLIMIT_FSIZE**.)

При выполнении ввода-вывода в отношении дискового файла успешный выход из **write()** не гарантирует перемещения данных на диск, поскольку ядро занимается буферизацией дискового ввода-вывода, чтобы сократить объем работы с диском и ускорить выполнение системного вызова **write()**.

## 3.6. Закрытие файла: close( )

Системный вызов **close()** закрывает открытый дескриптор файла, высвобождая его для последующего повторного использования процессом. Когда процесс прекращает работу, все его открытые дескрипторы файлов автоматически закрываются. Возвращает 0 при успешном завершении или -1 при ошибке:

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

Обычно предпочтительнее явно закрывать ненужные дескрипторы файлов. Тогда код, с учетом последующих изменений, будет проще для чтения и надежнее. Более того, дескрипторы файлов являются расходуемым ресурсом, поэтому сбой при закрытии дескриптора файла может вылиться в исчерпание процессом ограничения дескрипторов. Это, в частности, играет важную роль при написании программ, рассчитанных на долговременную работу и обращающихся к большому количеству файлов, например при создании оболочек или сетевых серверов.

Как и любые другие системные вызовы, **close()** должен сопровождаться проверкой кода на ошибки:

```
if (close (fd) == -1 ) exit (-1);
```

Такой код отлавливает ошибки вроде попыток закрытия неоткрытого дескриптора файла или закрытия одного и того же дескриптора файла дважды.

## 3.7. Изменение файлового смещения: lseek( )

Для каждого открытого файла в ядре записывается *файловое смещение,* которое иногда также называют *смещением чтения-записи* или *указателем.* Оно обозначает место в файле, откуда будет стартовать работа следующего системного вызова **read()** или **write()**.

Файловое смещение выражается в виде обычной байтовой позиции относительно начала файла. Первый байт файла расположен со смещением 0. При открытии файла смещение устанавливается на его начало, а затем автоматически корректируется каждым последующим вызовом **read()** или **write()**, чтобы указывать на следующий байт файла непосредственно после считанного или записанного байта (или байтов). Таким образом, успешно проведенные вызовы **read()** и **write()** идут по файлу последовательно.

Системный вызов **lseek()** устанавливает файловое смещение открытого файла, на который указывает дескриптор **fd**, в соответствии со значениями, заданными в аргументах **offset** и **whence**. Возвращает новое файловое смещение при успешном завершении или –1 при ошибке:

```
#include <unistd.h>
off_t lseek (int fd, off_t offset, int whence);
```

Аргумент **offset** определяет значение смещения в байтах. (Тип данных **off_t** – целочисленный тип со знаком, определенный в SUSv3.) Аргумент **whence** указывает на отправную точку, от которой отсчитывается смещение, и может иметь следующие значения:

- **SEEK_SET** – файловое смещение устанавливается в байтах на расстоянии **offset** от начала файла;
- **SEEK_CUR** – смещение устанавливается в байтах на расстоянии **offset** относительно текущего файлового смещения;
- **SEEK_END** – файловое смещение устанавливается на размер файла плюс **offset**.

Иными словами, **offset** рассчитывается относительно следующего байта после последнего байта файла.

Порядок интерпретации аргумента **whence** показан на рис. 3.1.

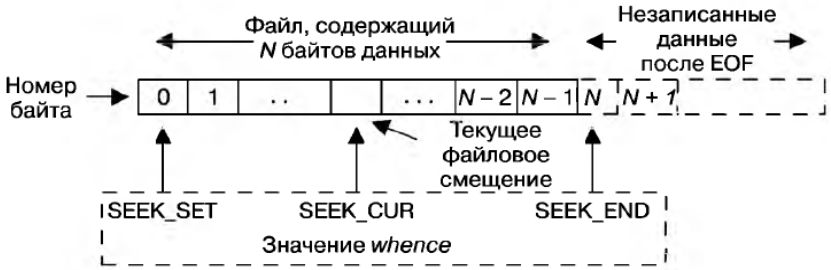



*Рис. 3.1.* Интерпретация аргумента **whence** системного вызова **lseek()**

Если аргумент **whence** содержит значение **SEEK_CUR** или **SEEK_END**, то у аргумента **offset** может быть положительное или отрицательное значение. Для **SEEK_SET** значение **offset** должно быть неотрицательным.

При успешном выполнении **lseek()** возвращается значение нового файлового смещения.

Чтобы извлечь текущее расположение файлового смещения, не изменяя его значения, используется следующий вызов:

```
curr = lseek (fd, 0, SEEK_CUR);
```

Рассмотрим некоторые другие примеры вызовов **lseek()**, а также комментарии, объясняющие, куда передвигается файловое смещение:

```
lseek (fd, 0, SEEK_SET); /* Начало файла * /
```

```
lseek (fd, 0, SEEK_END); /* Следующий байт после
конца файла */
lseek (fd, -1, SEEK_END); /* Последний байт файла * /
lseek (fd, -10, SEEK_CUR); /* Десять байтов
до текущего размещения */
lseek (fd, 10000, SEEK_END); /* 10 000 и 1 байт
после последнего байта файла */
```

Вызов **lseek()** просто устанавливает значение для записи ядра, содержащей файловое смещение и связанной с дескриптором файла. Никакого физического доступа к устройству при этом не происходит.

Не ко всем типам файлов можно применять системный вызов **lseek()**. Запрещено применение **lseek()** к программному каналу, каналу FIFO, сокету или терминалу – вызов аварийно завершится с установленным для **errno** значением **ESPIPE**. С другой стороны, **lseek()** можно применять к тем устройствам, в отношении которых есть смысл это делать, например при наличии возможности установки на конкретное место на дисковом или ленточном устройстве.

Буква «*l*» в названии **lseek()** появилась из-за того, что как для аргумента **offset**, так и для возвращаемого значения первоначально определялся тип **long**. В ранних реализациях UNIX предоставлялся системный вызов **seek()**, в котором для этих значений определялся тип **int**.

## 3.8. Блокировка доступа к файлу

Так же как замок на двери предотвращает нежелательные проникновения в дом, *блокировка* файла предотвращает доступ к данным в файле. Имеются следующие аспекты блокировки файлов:

1. *Блокировка записи*. Является блокировкой части файла. Поскольку файлы UNIX являются потоками байтов, было бы корректнее использовать термин *блокировка диапазона* (*range lock*), поскольку осуществляется блокировка диапазона байтов. Тем не менее термин «блокировка записей» общеупотребительный. Предоставляет исключительный доступ к записываемой области. Если эта область заблокирована также и для чтения, попытка получения блокировки записи либо блокируется, либо завершается неудачей. После получения блокировки записи попытка получить блокировку чтения завершается неудачей.

2. *Блокировка всего файла*. Как предполагает название, блокирует весь файл, даже если его размер меняется в блокированном состоянии. Интерфейс BSD предусматривает блокирование лишь всего файла. Для блокирования всего файла с использованием интерфейса POSIX указывают нулевую длину. Это интерпретируется особым образом – как «весь файл».

3. *Блокировка чтения.* Предотвращает запись в читаемую область. В файле может быть несколько заблокированных для чтения участков, даже в одной области файла, не мешающих друг другу, поскольку к данным осуществляется лишь доступ и они не изменяются.

### 3.8.1. Описание блокировки

Прежде чем рассмотреть осуществление блокировки, давайте исследуем описание блокировки в операционной системе. Это делается при помощи структуры **struct flock**, которая описывает диапазон блокируемых байтов и вид нужной блокировки:

```
struct flock {
short l_type;  // Тип: F_RDLCK, F_WRLCK, F_UNLCK
short l_whence;//SEEK_SET, SEEK_CUR, SEEK_END
off_t l_start; /* блокируемое смещение */
off_t l_len;   /* Число блокируемых байтов;
//0 означает от начала до конца файла
pid_t l_pid; /* PID блокирующего процесса, только
для F_GETLK
};
```

Поле **l_start** является смещением начального байта блокируемого участка. **l_len** является длиной блокируемого участка, т. е. общим числом блокируемых байтов. **l_whence** указывает место в файле, относительно которого отсчитывается **l_start**, значения те же, что и для аргумента **whence** функции **lseek()**.

Эта структура самодостаточна: смещение **l_start** и значение **l_whence** *не* связаны с текущим файловым указателем для чтения или записи. Пример кода мог бы выглядеть таким образом:

```
struct employee { /* что угодно */ };
struct flock lock; /* Структура блока */
lock.l_whence = SEEK_SET; /* Абсолютное положение */
```

```
lock.l_start = 5 * sizeof(struct employee);
/* Начало 6-й структуры */
lock.l_len = sizeof(struct employee);/*Блокировать
одну запись*/
```

Используя **SEEK_CUR** или **SEEK_END**, можно заблокировать участки, начиная от текущего смещения в файле или относительно конца файла соответственно. Для этих двух случаев **l_start** может быть отрицательным, пока абсолютное начало не меньше нуля.

Таким образом, чтобы заблокировать последнюю запись в файле:

```
lock.l_whence = SEEK_END; // Относительно EOF
lock.l_start = -1 * sizeof (struct employee);
/* Начало последней структуры */
lock.l_len = sizeof(struct employee);
```

Установка **l_len** в 0 является особым случаем. Он означает блокировку файла от начального положения, указанного с помощью **l_start** и **l_whence**, и до конца файла. Сюда входят также любые области за концом файла. (Если заблокированный файл увеличивается в размере, область блокировки расширяется так, чтобы продолжать охватывать весь файл.) Таким образом, блокирование всего файла является вырожденным случаем блокирования одной записи:

```
lock.l_whence = SEEK_SET; /* Абсолютное положение */
lock.l_start = 0; /* Начало файла */
lock.l_len = 0; /* До конца файла */
```

Теперь, когда мы знаем, как описать, *где* блокируется файл, мы можем описать *тип* блокировки с помощью **l_type**. Возможные следующие значения:

- **F_RDLCK**: блокировка чтения. Для применения блокировки чтения файл должен быть открыт для чтения;
- **F_WRLCK**: блокировка записи. Для применения блокировки записи файл должен быть открыт для записи;
- **F_UNLCK**: освобождение предыдущей блокировки.

Таким образом, полная спецификация блокировки включает установку в структуре **struct flock** значений четырех полей: трех для указания блокируемой области и четвертого для описания нужного типа блока.

### 3.8.2. Блокировка функцией fcntl( )

После заполнения структуры **struct flock** следующим шагом является запрос блокировки с помощью соответствующего значения аргумента **cmd** функции **fcntl()**:

- **F_GETLK**: узнать, можно ли установить блокировку;
- **F_SETLK**: установить или снять блокировку;
- **F_SETLKW**: установить блокировку, подождав, когда это станет возможным.

Команда **F_GETLK** осведомляется, доступна ли описанная **struct flock** блокировка. Если она доступна, блокировка *не* устанавливается, вместо этого операционная система изменяет поле **l_type** на **F_UNLCK**. Другие поля остаются без изменений. Если блокировка недоступна, операционная система заполняет различные поля сведениями, описывающими уже установленные блокировки, которые препятствуют установке новой. Если блокировка уже установлена, нет другого выбора, кроме ожидания в течение некоторого времени и новой попытки установки блокировки или вывода сообщения об ошибке и отказа от дальнейших попыток.

Команда **F_SETLK** пытается установить указанную блокировку. Если **fcntl()** возвращает 0, блокировка была успешно установлена. Если она возвращает –1, блокировку установил другой процесс. В этом случае в **errno** устанавливается **EAGAIN** или **EACCESS**.

Команда **F_SETLKW** также пытается установить указанную блокировку. Она отличается от **F_SETLK** тем, что будет ждать, пока установка блокировки не окажется возможной.

Выбрав соответствующее значение для аргумента **cmd**, передайте его в качестве второго аргумента **fcntl()** вместе с указателем на заполненную структуру **struct flock** в качестве третьего аргумента:

```
struct flock lock;
int fd;
/* ...открыть файл, заполнить struct flock... */
if (fcntl(fd, F_SETLK, &lock) < 0) {
/* Установить не удалось, попытаться восстановиться */
}
```

Завершив работу с заблокированным участком, его следует освободить. Для этого возьмите первоначальную **struct lock**, исполь-

зованную для блокирования, и измените поле **l_type** на **F_UNLCK**. Затем используйте **F_SETLK** в качестве аргумента **cmd**:

```
lock.l_type = F_UNLCK; /* Разблокировать */
if (fcntl(fd, F_SETLK, &lock) < 0) {
/* обработать ошибку */
}
/* Блокировка была снята */
```

### 3.8.2. Блокировка функцией lockf( )

Функция **lockf()** предоставляет альтернативный способ установки блокировки *в текущем положении файла*.

```
#include <sys/file.h>
int lockf(int fd, int cmd, off_t len);
```

Дескриптор файла **fd** должен быть открыт для записи. **len** указывает число блокируемых байтов: от текущего положения (назовем его **pos**) до **pos + len** байтов, если **len** положительно, или от **pos - len** до **pos - 1**, если **len** отрицательно.

Команды (**cmd**) следующие:

- **F_LOCK**: устанавливает исключительную блокировку диапазона. Вызов блокируется до тех пор, пока блокировка диапазона не станет возможной;
- **F_TLOCK**: пытается установить блокировку. Это похоже на **F_LOCK**, но если блокировка недоступна, то **F_TLOCK** возвращает ошибку;
- **F_ULOCK**: разблокирует указанный раздел;
- **F_TEST**: проверяет, доступна ли блокировка. Если доступна, возвращает 0 и устанавливает блокировку. В противном случае возвращает –1 и устанавливает в **errno** **EACCESS**.

Возвращаемое значение равно 0 в случае успеха и –1 при ошибке, с соответствующим значением в **errno**. Возможные значения ошибок включают:

- **EAGAIN**: файл заблокирован, для **F_TLOCK** или **F_TEST**;
- **ENOLCK**: операционная система не смогла выделить блок.

Завершив работу с заблокированным участком, его следует освободить. Код, использующий **lockf()**, несколько проще, чем для **fcntl()**. Для краткости опустим проверку ошибок:

```
off_t curpos, len;
curpos = lseek(fd, (off_t)0, SEEK_CUR);
len = ... ; / * Число блокируемых байтов */
lockf(fd, F_LOCK, len); / * Блокировка */
/* использование заблокированного участка... */
lseek(fd, curpos, SEEK_SET); /* Вернуться к началу
блокировки */
lockf(fd, F_ULOCK, len); // Разблокировать файл
```

Если вы не освободите блокировку явным образом, операционная система сделает это за вас в двух случаях. Первый случай – когда процесс завершается (либо при возвращении из **main()** либо с использованием функции **exit()**). Другим случаем является вызов **close()** с дескриптором файла.

Вызов **close()** с *любым* открытым для файла дескриптором удаляет *все* блокировки файла процессом, даже если другие дескрипторы для файла остаются открытыми.

Вызовы функций блокировки POSIX ***не следует*** использовать в сочетании с библиотекой **<stdio.h>**. Эта библиотека осуществляет свое собственное буферирование. Хотя можно получить с помощью **fileno (FLIE *)** дескриптор нижележащего файла, действительное положение в файле может быть не там, где вы думаете. В общем, стандартная библиотека ввода-вывода не понимает блокировок файлов средствами системных вызовов. Однако существует семейство функций, которые не представляют для нас интереса, поскольку ничего не делают, если блокировка файла осуществляется функциями типа **fcntl()** или **lockf()**:

```
include <stdio.h>
void flockfile(FILE *filehandle);
int ftrylockfile(FILE *filehandle);
void funlockfile(FILE *filehandle);
```

## Вопросы для самопроверки

1. Перечислите стандартные дескрипторы файлов. Чем они отличаются от дескрипторов обычных файлов?
2. В чем заключается универсальность модели ввода-вывода UNIX?
3. В чем отличия вызова функции **`open()`** для создания нового файла и открытия существующего?
4. Перечислите форматы и значения третьего аргумента функции **`open()`**.
5. Перечислите дополнительные (кроме режима доступа) флаги функции **`open()`**.
6. Каковы основные ошибки, могущие возникнуть при открытии файла?
7. Каковы особенности работы функции **`read()`**?
8. Каковы особенности работы функции **`write()`**?
9. Почему нужно явно вызывать функцию **`close()`**?
10. Для чего служит функция **`lseek()`**?
11. Каковы допустимые аргументы функции **`lseek()`**?
12. К каким типам файлов нельзя применять функцию **`lseek()`**?
13. Какие виды блокировки файла существуют в LINUX?
14. Приведите описание структуры блокировки файла, опишите ее поля.
15. Опишите процесс блокировки файла с помощью функции **`fcntl()`**.
16. Опишите процесс блокировки файла с помощью функции **`lockf()`**.
17. Чем отличается применение функций блокировки **`fcntl()`** и **`lockf()`**?
18. В каких случаях блокировка файла снимается операционной системой?
19. Возможно ли сочетать функции блокировки POSIX с файловыми функциями стандартной библиотеки C (**`<stdio.h>`**)? Почему?
20. Каковы особенности функций блокировки файлов средствами стандартной библиотеки C (**`<stdio.h>`**)?

## Упражнения

Выполните лабораторную работу № 1 из лабораторного практикума.

# Глава 4. Статические и динамические библиотеки

## 4.1. Библиотека объектов

Обычно простые программы состоят из одного исходного файла. Если программа становится большой, то при работе с ней может возникнуть несколько достаточно серьезных проблем:

- файл, становясь большим, увеличивает время компиляции, и малейшие изменения в исходном тексте автоматически вынуждают тратить время программиста на перекомпиляцию программы;
- если над программой работает много человек, то практически невозможно отследить сделанные изменения;
- процесс правки и само ориентирование при большом исходном тексте становится сложным, и поиск небольшой ошибки может повлечь за собой вынужденное «изучение» кода заново.

Поэтому при разработке программ рекомендуется разбивать их на куски, которые функционально ограничены и закончены. Для того чтобы вынести функцию или переменную в отдельный файл, надо перед ней поставить зарезервированное слово **extern**. Для примера создадим программу из нескольких файлов. Сначала создадим главную программу, в которой будут две внешние процедуры. Назовем этот файл main0.c (см. [6], «Примеры программ», «Библиотеки»):

```
#include <stdio.h>
extern int f1();
extern int f2();
int main() {
int n1, n2;
```

```
n1 = f1(); n2 = f2();
printf("f1() = %d\n",n1); printf("f2() = %d\n",n2);
return 0;
}
```

Теперь создаем два файла (см. [6]), каждый из которых будет содержать полное определение внешней функции из главной программы. Файлы назовем f1.c и f2.c:

```
// файл f1.c
int f1() { return 2; }
// файл f2.c
int f2() { return 10; }
```

Компилировать можно все файлы одновременно одной командой, перечисляя составные файлы через пробел после ключа **-c**:

```
gcc -c main0.c f1.c f2.c
```

В результате работы компилятора мы получим три отдельных объектных файла:

main0.o, f1.o, f2.o

Чтобы их собрать в один файл с помощью **gcc**, надо использовать ключ **-o**:

```
gcc main0.o f1.o f2.o -o rezult
```

В результате вызова полученной программы **rezult** командой:

```
./rezult
```

на экране появится результат работы:

```
[gun]$./rezult f1() = 2 f2() = 10
```

Теперь при изменении какой-либо из процедур, например **f1()**:

```
int f1() { return 25; }
```

компилировать заново все файлы не придется, а понадобится лишь скомпилировать измененный файл и собрать результирующий файл из кусков.

Таким образом, можно создавать большие проекты, которые больше не будут отнимать много времени на компиляцию и поиск ошибок.

На самом деле сборка выполняется отдельной программой-компо-новщиком **ld**. Когда мы компилируем программу с помощью команды **gcc**, утилита **ld** вызывается автоматически, незаметно для нас.

Библиотека объектных файлов – это файл, содержащий несколько объектных файлов, которые будут использоваться вместе на стадии сборки (связывания, линковки) программы. Нормальная библиотека содержит символьный индекс, состоящий из названий функций, переменных и т. д., которые содержатся в библиотеке. Это позволяет ускорить процесс сборки программы. Библиотеки объектов бывают двух видов: статические и динамические (разделяемые).

***Статическая библиотека*** – это коллекция объектных файлов, которые присоединяются к программе во время сборки программы. Таким образом, статические библиотеки используются только при создании программы. Потом в работе самой программы они не принимают участия, в отличие от динамических библиотек.

***Динамическая библиотека*** – это созданная специальным образом библиотека, которая присоединяется к результирующей программе в два этапа. Первый этап – это, естественно, этап компиляции. На этом этапе компилятор встраивает в программу описания требуемых функций и переменных, которые присутствуют в библиотеке. Сами объектные файлы из библиотеки не присоединяются к программе.

Присоединение этих объектных файлов (кодов функций) осуществляет системный динамический загрузчик во время запуска программы. Загрузчик проверяет все библиотеки, связанные с программой, на наличие требуемых объектных файлов, затем загружает их в память и присоединяет к копии запущенной программы, находящейся в памяти.

Сложный процесс загрузки динамических библиотек замедляет запуск программы, но у него есть существенный плюс: если другая запускаемая программа связана с этой же загруженной динамической библиотекой, то она использует ту же копию библиотеки. Это означает, что требуется гораздо меньше памяти для запуска нескольких программ, а сами загрузочные файлы меньше по размеру, что экономит место на дисках.

## 4.2. Статические библиотеки

Чтобы лучше понимать особенности и преимущества разделяемых библиотек, вначале кратко рассмотрим их статические аналоги. Статические библиотеки (также известные как архивы) были первым видом библиотечных файлов, доступных в UNIX-системах. Они имеют следующие положительные стороны:

- можно поместить набор часто используемых объектных файлов в единую библиотеку, которую потом можно будет применять для сборки разных программ; при этом не нужно будет перекомпилировать оригинальные исходные тексты при компоновке каждой программы;
- упрощаются команды для компоновки. Вместо перечисления длинного списка объектных файлов можно указать всего лишь имя статической библиотеки. Компоновщик знает, как выполнять поиск по ней и извлекать объекты, необходимые для создания исполняемого файла.

Статическая библиотека, по сути, является обычным файлом, содержащим копии всех помещенных в него объектных файлов. Статическим библиотекам принято давать имена вида libname.a.

### 4.2.1 Создание и редактирование статической библиотеки

Для создания и редактирования статических библиотек используется команда **ar,** которая имеет следующую общую форму:

```
$ ar options archive object-file...
```

Аргумент **options** состоит из набора букв, одна из которых является *кодом операции,* а остальные – *модификаторами,* влияющим на то, как эта операция будет выполняться. Ниже приведена часть распространенных кодов операции:

- операция **c** заставляет создавать (от англ. create) библиотеку, если ее нет;
- **r** (от англ. replace – «заменить»). Вставляет объектный файл в архив, заменяя им любой существующий файл с тем же именем. Это стандартный способ создания и обновления архивов. Таким образом, архив можно собрать с помощью следующих команд:

```
$ gcc  -c mod1.c mod2.c mod3.c
$ ar cr libdemo.a mod1.o mod2.o mod3.o
$ rm mod1.o mod2.o mod3.o
```

• **t** (от англ. table of contents – «оглавление»). Выводит оглавление архива. По умолчанию выводятся только имена объектных файлов. Но если дополнительно указать параметр **v** (от англ. verbose – «подробно»), можно просмотреть все атрибуты каждого файла в архиве, как показано ниже:

```
$ ar tv libdemo.a
rw- r - - r - - 1000/100 1001016 Nov 15 12:26 2019
mod1.o
rw- r - - r - - 1000/100 406668 Nov 15 12:21 2019
mod2.o
rw- r - - r - - 1000/100 46672 Nov 15 12:21 2019
mod3.o
```

Дополнительные атрибуты каждого объекта слева направо: права доступа на момент добавления в архив, пользовательский и групповой идентификаторы, размер, а также дата и время последнего изменения;

• **d** (от англ. delete – «удалить»). Удаляет из архива заданный модуль, как показано в следующем примере:

```
$ ar d libdemo.a mod3.o
```

Пока у нас есть лишь архивный файл **libdemo.a**. Чтобы из него сделать полноценную библиотеку объектных файлов, надо добавить к этому архиву индекс символов, т. е. список вложенных в библиотеку функций и переменных, чтобы линковка происходила быстрее. Индексация выполняется утилитой **ranlib**:

```
ranlib libимя_библиотеки.a
```

### 4.2.2. Использование статической библиотеки

Скомпоновать программу со статической библиотекой можно двумя способами. Первый из них таков: название файла библиотеки можно указать на этапе компоновки, как показано ниже:

```
$ gcc -c prog.c
$ gcc -o prog prog.o libdemo.a
```

Можно также поместить библиотеку в один из стандартных каталогов, по которым компоновщик выполняет поиск (например,

**/usr/lib**), и затем указать ее имя (то есть имя файла без префикса lib и суффикса .a) с помощью параметра **-l**:

```
$ gcc -o prog prog.o -ldemo
```

Если библиотека находится в каталоге, о котором компоновщику обычно ничего не известно, можно воспользоваться параметром **-L**, чтобы указать этот каталог отдельно:

```
$ gcc -o prog prog.o -Lmylibdir -ldemo
```

Статическая библиотека может состоять из множества объектных модулей, но компоновщик выберет из них только те, которые нужны программе.

## 4.3. Краткий обзор разделяемых библиотек

Когда программа компонуется со статической библиотекой (или вовсе без использования библиотек), итоговый исполняемый файл содержит копии всех объектных модулей, скомпонованных с программой. Таким образом, несколько разных программ могут содержать в себе копии одних и тех же объектных модулей (функций) (рис. 4.1.).

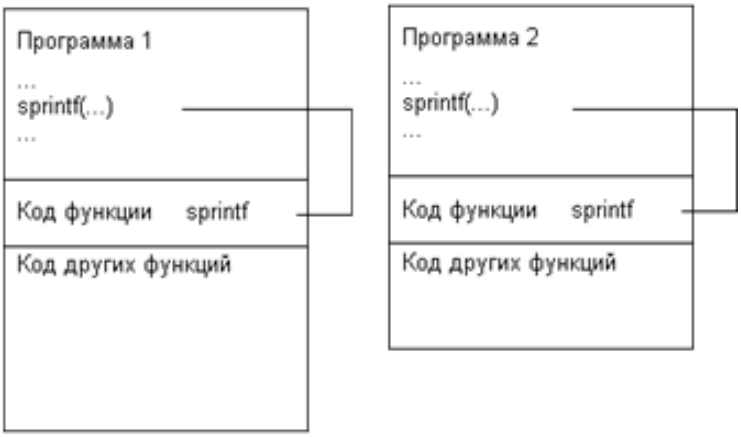



*Рис. 4.1.* Вызов функций при использовании статической компоновки

Подобная избыточность несет в себе несколько недостатков:

- дисковое пространство уходит на хранение нескольких копий одних и тех же объектных модулей. Такие потери могут быть значительными;
- если несколько программ, применяющих одни и те же модули, выполняются одновременно, то каждая из них будет хранить в виртуальной памяти свою отдельную копию этих модулей, увеличивая тем самым потребление виртуальной памяти в системе;
- если объектный модуль статической библиотеки требует каких-либо изменений (возможно, нужно исправить ошибку), придется заново компоновать все исполняемые файлы, в которых этот модуль используется.

Для устранения представленных недочетов были придуманы разделяемые библиотеки. Их ключевая идея состоит в том, что одна копия объектного модуля разделяется между всеми программами, использующими его. Объектные модули не копируются в компонуемый исполняемый файл, вместо этого единая копия библиотеки загружается в память при запуске первой программы, которой требуются ее объектные модули. Если позже будут запущены другие программы, использующие эту разделяемую библиотеку, они обращаются к копии, уже загруженной в память (рис. 4.2). Благодаря применению разделяемых библиотек исполняемые файлы требуют меньше места на диске и в виртуальной памяти (при выполнении).

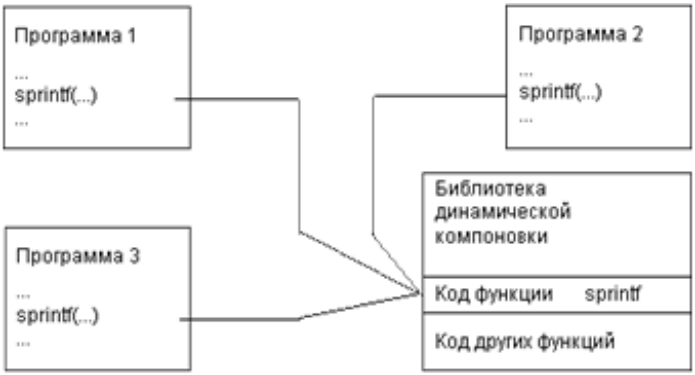



*Рис. 4.2.* Вызов функций при использовании динамической компоновки

Код разделяемых библиотек является общим для нескольких процессов, однако глобальные и статические переменные, объявленные внутри библиотеки, предоставляются в виде копий.

Кроме того, разделяемые библиотеки обладают следующими преимуществами.

- Общий размер программ уменьшается, в связи с чем в некоторых случаях они могут быстрее загружаться в память, что ускоряет их запуск. Это относится только к большим разделяемым библиотекам, которые уже используются другими программами. Программа, первой загружающая разделяемую библиотеку, запускается дольше, поскольку данную библиотеку сначала нужно найти и загрузить в память.
- Объектные модули не копируются в исполняемые файлы, а хранятся в единой разделяемой библиотеке, поэтому можно изменять общий код без необходимости выполнять повторную компоновку. Изменения можно вносить, даже когда запущенные программы уже применяют существующую версию разделяемой библиотеки.

Однако за эти дополнительные возможности приходится платить.

- Разделяемые библиотеки более сложные по сравнению со статическими – как с точки зрения самой концепции, так и на практике, при их создании и сборке программ, которые их используют.
- Разделяемые библиотеки должны быть скомпилированы с поддержкой адресно-независимого кода, который ввиду применения дополнительных регистров имеет издержки в большинстве архитектур.

## 4.4. Создание и использование разделяемых библиотек

### 4.4.1. Создание разделяемой библиотеки

Для построения разделяемой версии статической библиотеки, созданной ранее, нужно выполнить следующие шаги:

```
$ gcc -g -c -fPIC -Wall mod1.c mod2.c mod3.c
$ gcc -g -shared -o libfoo.so mod1.o mod2.o mod3.o
```

Первая команда создает три объектных модуля (назначение параметра **`gcc -fPIC`** разъясним далее). Команда **`gcc -shared`** создает на основе этих модулей разделяемую библиотеку.

Имена разделяемых библиотек принято обрамлять префиксом **lib** и суффиксом **.so** (от англ. shared object – «разделяемый (совместно используемый) объект»). Стоит также отметить, что скомпилировать исходные файлы и создать разделяемую библиотеку можно с помощью всего одной команды:

```
$ gcc -fPIC -Wall mod1.c mod2.c mod3.c -shared -o libfoo.so
```

В отличие от статических библиотек готовые разделяемые библиотеки не позволяют добавлять или удалять отдельные объектные модули. Как и в случае с обычными исполняемыми программами, объектные файлы внутри разделяемой библиотеки теряют свою идентичность.

### 4.4.2. Адресно-независимый код

Параметр **gcc -fPIC** заставляет компилятор сгенерировать *адресно-независимый код.* Это влияет на такие операции, как доступ к глобальным, статическим и внешним переменным, строковым константам, а также на то, как определяются адреса функций. Данные изменения позволяют размещать исполняемый код на любом участке виртуальной памяти. Такой механизм является необходимым для разделяемых библиотек, поскольку на этапе компоновки невозможно определить, куда именно будет загружен их код (точное местоположение разделяемой библиотеки в адресном пространстве зависит от различных факторов, таких как количество памяти, которое уже использует загружающая библиотеку программа, и какие разделяемые библиотеки она успела загрузить).

### 4.4.3. Использование разделяемой библиотеки

Перед применением разделяемой библиотеки необходимо выполнить два шага, которые не требуются для работы со статическими библиотеками.

1. Поскольку исполняемый файл больше не содержит копии нужных ему объектных модулей, он должен иметь возможность определять, какая разделяемая библиотека требуется для его работы. Для этого на этапе компоновки в исполняемый файл внедряется имя библиотеки. Набор всех разделяемых библиотек, которые нужны программе, называют *списком динамических зависимостей.*

2. Должен существовать механизм для поиска файла библиотеки по ее имени во время выполнения программы и последующей его загрузки в память, если он не был загружен до этого.

Внедрение имени библиотеки в исполняемый файл происходит автоматически при компоновке разделяемой библиотеки с программой:

```
$ gcc -Wall -o prog prog.c libfoo.so
```

Теперь, если запустить данную программу, можно будет увидеть следующее сообщение об ошибке:

```
$ ./prog
./prog: error in loading shared libraries:
libfoo.so: cannot open shared object file: No such
file or directory
```

Это подводит нас ко второму обязательному шагу – *динамической компоновке.* Данная процедура разрешения внедренного имени библиотеки на этапе выполнения, производимая *динамическим компоновщиком* (который еще называют *динамически компонуемым загрузчиком* или *компоновщиком времени выполнения*). Он сам по себе является разделяемой библиотекой, **/lib/ld-linux.so.2**, применяемой всеми исполняемыми файлами формата ELF (Executive Linux File, исполняемый файл Linux), использующими динамические библиотеки.

Путь **/lib/ld-linux.so.2** представляет собой обычную символьную ссылку на исполняемый файл динамического компоновщика.

Динамический компоновщик анализирует список динамических зависимостей программы и находит соответствующие библиотечные файлы, используя набор заранее заданных правил. Часть этих правил основывается на списке стандартных каталогов, в которых обычно хранятся разделяемые библиотеки (к примеру, **/lib** и **/usr/lib**). Причина сообщения об ошибке, приведенного выше, заключается в том, что библиотека находится в текущем каталоге, который не учитывается динамическим компоновщиком при выполнении поиска.

Для оповещения динамического компоновщика о том, что разделяемая библиотека находится в нестандартном месте, можно воспользоваться переменной среды **LD_LIBRARY_PATH**, указав соответствующий каталог в качестве одного из элементов списка, разделенного двоеточиями. Для разделения каталогов можно также использовать точку с запятой, но, чтобы избежать интерпретации данного символа командной оболочкой, список в этом случае следует обрамить кавычками.

Для начала стоит посмотреть, есть ли у нас такая переменная среды:

```
[gun]$ echo $LD_LIBRARY_PATH
```

У меня в ответ выводится пустая строка, означающая, что такой переменной среды нет. Устанавливается она следующим образом:

```
[gun]$ LD_LIBRARY_PATH=/home/gun/libs
[gun]$ export LD_LIBRARY_PATH
```

Если переменная **LD_LIBRARY_PATH** определена, компоновщик начинает поиск разделяемой библиотеки с тех каталогов, которые в ней перечислены, и только потом переходит к стандартным библиотечным путям. Следовательно, можно запустить программу и с помощью следующей команды:

```
$ LD_LIBRARY_PATH=. ./prog
```

Синтаксис командной оболочки (такой, как bash, ksh и sh), использованный выше, позволяет определить переменную среды в рамках выполнения программы **prog**. Это определение сообщает динамическому компоновщику о том, что поиск разделяемых библиотек нужно проводить в текущем каталоге.

Для настройки динамического компоновщика существует программа **ldd**. Она выдает на экран список динамических библиотек, используемых в программе, и их местоположение. В качестве параметра ей сообщается название обследуемой программы. **ldd** находит все модули, на которые ссылается библиотека (по тому же принципу, что и динамический компоновщик), и выводит результат в следующем виде:

```
имя_библиотеки => ссылается_на_путь
```

Для большинства исполняемых файлов в формате ELF команда **ldd** выведет:

```
[gun]$ ldd prog
libfoo.so => not found
libc.so.6 => /lib/libc.so.6 (0x40016000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Программа использует три библиотеки:

1. **libc.so.6** – стандартную библиотеку функций языка C.
2. **ld-linux.so.2** – библиотеку динамической компоновки программ формата ELF.
3. **libfoo.so** – нашу динамическую библиотеку функций.

Нашу библиотеку она найти не может. Динамический компоновщик ищет библиотеки только в известных ему каталогах. Для того чтобы добавить нашу директорию с библиотекой в список известных директорий, надо редактировать файл **/etc/ld.so.conf**. Наша библиотека станет «заметной», если поместить ее в один из перечисленных там каталогов либо дописать каталог в этот файл. После изменения конфигурационного файла **ld.so.conf** необходимо, чтобы система перечитала настройки заново. Это делает программа **ldconfig**. Но описанный метод влияет на всю систему в целом и требует доступа администратора системы, т. е. прав суперпользователя **root**.

Для простого пользователя есть другое решение. Это использование описанной ранее переменной среды **LD_LIBRARY_PATH**, в которой перечисляются все каталоги, содержащие пользовательские динамические библиотеки.

### 4.4.4. Команды objdump и readelf

Команда **objdump** позволяет получить из исполняемого файла, скомпилированного объекта или разделяемой библиотеки различную информацию, включая дизассемблированный машинный код в двоичном формате. С ее помощью также можно вывести содержимое заголовков разных ELF-разделов упомянутых файлов; в этом контексте она напоминает команду **readelf**, которая выводит похожие данные, но в другом формате. Подробное описание команд **objdump** и **readelf** см. в руководстве **man**.

### 4.4.5. Команда nm

Выводит список всех символов, определенных внутри объектной библиотеки или исполняемой программы. С ее помощью можно узнать, какой именно библиотеке принадлежит тот или иной символ. Например, чтобы понять, где определена функция **crypt()**, можно сделать следующее:

```
$ nm -A /usr/lib/lib *.so 2> /dev/null | grep 1
crypt$'
/usr/lib/libcrypt.so:00007080 W crypt
```

Параметр **-A** говорит о том, что перед каждым символом должно быть указано имя библиотеки. Он нужен, поскольку команда **nm** по

умолчанию сначала выводит имя библиотеки, а потом перечисляет все символы, которые ей принадлежат; это не подходит для выполнения фильтрации, как в нашем примере. Кроме того, мы отключили стандартный вывод ошибок (**> /dev/null**), чтобы скрыть сообщения о файлах, формат которых данная команда не поддерживает. На основе результата, полученного выше, можно сказать, что функция **crypt()** определена в библиотеке **libcrypt**.

### 4.4.6. Создание разделяемой библиотеки с применением общепринятых методик

Теперь, используя всю вышеприведенную информацию, мы покажем, как создать демонстрационную библиотеку, следуя общепринятым методикам. Для начала создадим объектные файлы:

```
$ gcc -c -fPIC -Wall mod1.c mod2.c mod3.c
```

Затем построим разделяемую библиотеку с именем **libdemo.so**:

```
$ gcc -shared libdemo.so mod1.o mod2.o mod3.o
```

Построим наш исполняемый файл, применяя компоновочное имя, и запустим программу уже привычным нам способом:

```
$ gcc -g -Wall -o prog prog.c -L. -ldemo
$ LD_LIBRARY_PATH=. ./prog
```

Если в распоряжении компоновщика имеются оба типа библиотеки с тем же именем – статическая и разделяемая (например, когда во время компоновки указаны параметры **-Lsomedir -ldemo**, и при этом существуют файлы libdemo.so и libdemo.a), то он по умолчанию выбирает динамическую. Чтобы заставить его использовать статическую версию библиотеки, можно выполнить одно из следующих действий:

- указать путь к статической библиотеке (включая расширение **.a**) в командной строке **gcc**;
- указать для **gcc** параметр **-static**;
- задействовать параметры **gcc –W1**, **-Bstatic**, чтобы явно переключить компоновщик в режим использования статических библиотек. При этом можно применять параметры **-l** в командной строке **gcc**. Компоновщик обработает их в том порядке, в котором они были указаны.

## 4.5. Динамически загружаемые библиотеки

При запуске исполняемого файла динамический компоновщик загружает все разделяемые библиотеки, указанные в списке динамических зависимостей программы. Но иногда может понадобиться загрузить библиотеку чуть позже. Например, подключаемый модуль загружается, только когда нужен. Эта возможность предоставляется программным интерфейсом динамического компоновщика. Данный интерфейс обычно называют **dlopen**, изначально он был разработан для системы Solaris, но теперь большая его часть описана в стандарте SUSv3.

Интерфейс **dlopen** позволяет программе открыть разделяемую библиотеку во время выполнения, найти в ней функцию с подходящим именем и вызвать ее. Библиотеки, которые используются таким образом, обычно называют *динамически загружаемыми;* при этом они создаются точно так же, как и любые другие разделяемые библиотеки.

Основу **dlopen** составляют представленные ниже функции (все они входят в стандарт SUSv3):

- **dlopen()** – открывает разделяемую библиотеку и возвращает дескриптор, который можно использовать в последующих вызовах;
- **dlsym()** – ищет в библиотеке определенный символ (строку, содержащую имя функции или переменной) и возвращает его адрес;
- **dlclose()** – закрывает библиотеку, открытую ранее вызовом **dlopen()**;
- **dlerror()** – возвращает строку с сообщением об ошибке и применяется после неудачного завершения одной из вышеописанных функций.

Чтобы собрать в Linux программу, использующую программный интерфейс **dlopen**, нужно указать параметр **-ldl**, это позволит скомпоновать ее с библиотекой **libdl**.

### 4.5.1. Открытие разделяемой библиотеки: dlopen( )

Функция **dlopen()** загружает в виртуальное адресное пространство вызывающего ее процесса разделяемую библиотеку с именем **libfilename** и инкрементирует счетчик открытых ссылок на нее.

Возвращает дескриптор библиотеки при успешном завершении или **NULL** при ошибке:

```
#include <dlfcn.h>
void *dlopen(const char *libfilename, int flags);
```

Если имя **libfilename** содержит слеш (/), то **dlopen()** интерпретирует его как относительный или полный путь. В противном случае динамический компоновщик ищет разделяемую библиотеку по ранее описанному принципу.

В случае успеха **dlopen()** возвращает дескриптор, по которому можно ссылаться на библиотеку в последующих вызовах функций из программного интерфейса **dlopen**. Если произойдет ошибка (например, библиотеку не удалось найти), то **dlopen()** возвращает **NULL**. В таком случае тест ошибки, понятный человеку, можно получить с помощью функции **dlerror()**.

Если разделяемая библиотека, указанная с помощью аргумента **libfilename**, зависит от других библиотек, то **dlopen()** загрузит их автоматически. При необходимости эта процедура выполняется рекурсивно. Мы будем называть набор загруженных таким образом библиотек *деревом зависимостей.*

Функцию **dlopen()** можно вызвать несколько раз для одной и той же библиотеки. При этом загрузка будет выполнена лишь при первом вызове, а во всех последующих случаях станет возвращаться одно и то же значение **handle**. Однако программный интерфейс **dlopen** хранит счетчик ссылок для каждого дескриптора. С каждым вызовом **dlopen()** он инкрементируется, а декрементация происходит при вызове **dlclose()**; последний выгружает библиотеку из памяти только в том случае, если счетчик равен 0.

Аргумент **flags** представляет собой битовую маску, значение которой должно быть равно либо **RTLD_LAZY**, либо **RTLD_NOW**. Эти константы описаны ниже.

**RTLD_LAZY** – неопределенные ссылки на функции библиотеки должны быть разрешены только при выполнении соответствующего кода. Если участок кода, которому требуется определенный символ, не выполняется, то данный символ не разрешается. Отложенное разрешение производится только для ссылок на функции; ссылки на переменные всегда разрешаются незамедлительно. Наличие флага **RTLD_LAZY** обеспечивает поведение, соответствующее обычной работе динамиче-

ского компоновщика, когда тот загружает разделяемые библиотеки из списка динамических зависимостей исполняемого файла.

**RTLD_NOW** – все неопределенные ссылки библиотеки должны быть немедленно разрешены вне зависимости от того, понадобятся ли они когда-нибудь; данные операции нужно произвести до завершения вызова **dlopen()**. Это замедляет загрузку библиотеки, но позволяет сразу же определить все ошибки, связанные со ссылками на функции. Такой подход может оказаться полезным при отладке приложения или в случае, когда при обнаружении неразрешенного символа программа должна завершиться немедленно, а не в ходе дальнейшего выполнения.

Приведем пример кода для открытия библиотеки:

```
void *library_handler;
library_handler = dlopen("/path/to/the/library.so",
RTLD_LAZY);
if (!library_handler){ //если ошибка
fprintf(stderr,"dlopen() error: %s\n", dlerror());
exit(1);
};
```

### 4.5.2. Получение адреса функции или переменной: dlsym( )

После открытия библиотеки можно получить адрес функции или переменной в ней по имени с помощью функции:

```
void *dlsym(void *handle, char *symbol);
```

Для этой функции требуется адрес загруженной библиотеки **handle**, полученный при открытии функцией **dlopen()**. Требуемая функция или переменная задается своим именем в переменной **symbol**.

Если аргумент **symbol** содержит имя функции, то эту функцию можно вызвать с помощью указателя, полученного в результате вызова **dlsym()**. Результат выполнения **dlsym()** можно поместить в заранее определенный указатель подходящего типа, как показано ниже:

```
int ( *funcp ) ( int ) ; /* Указатель на функцию,
принимающую целочисленный аргумент и возвращающую
целочисленный результат */
```

Однако мы не можем просто присвоить результат выполнения **dlsym()** такому указателю, как показано ниже:

```
funcp = dlsym (handle, symbol);
```

Причина в том, что стандарт С99 запрещает операцию присваивания между указателем на функцию и **void *** . В качестве решения можно воспользоваться (немного грубым) приведением типов:

```
* ( void ** ) (&funcp ) = dlsym (handle, symbol);
```

Получив указатель на функцию с помощью **dlsym()**, можно вызвать ее путем обычной для языка С операции разыменовывания:

```
res = ( *funcp ) (somearg);
```

### 4.5.3. Выгрузка динамической библиотеки: dlclose( )

Выгружается библиотека функцией:

```
dlclose(void *handle);
```

При закрытии библиотеки динамический компоновщик проверяет счетчик количества открытий библиотеки, и если она была открыта несколькими программами одновременно, то она не выгружается до тех пор, пока все программы не закроют эту библиотеку.

### 4.5.4. Пример применения

Для примера создадим программу, где в качестве параметров – названия функций, которые она будет использовать в работе. Например, математические функции возведения в степень. Напишем текст функций:

```
double power2(double x){return x*x;};
double power3(double x){ return x*x*x; };
```

Сохраняем его в файл lib.c (см. [6]) и создаем динамическую библиотеку **libpowers.so** следующими командами:

```
[gun]$ gcc -fPIC -c lib.c
[gun]$ gcc -shared lib.o -o libpowers.so
```

Теперь пишем использующую ее основную программу в файле main1.c (см. [6]):

```
#include <stdio.h>
#include <dlfcn.h>
int main(int argc, char* argv[]){
void *ext_library;
double value=0;
double (*powerfunc)(double x); // Прототип функции
ext_library = dlopen ("/gun/libs/libpowers.so",
RTLD_LAZY);
if (!ext_library){
fprintf(stderr,"dlopen() error: %s\n", dlerror());
return 1; };
powerfunc = dlsym(ext_library, argv[1]); //Ошибка,
если -std=C99
value=3.0;
// Вызов функции по адресу
printf("%s(%f) = %f\n",argv[1],value,(*powerfunc)
(value));
dlclose(ext_library); };
```

Код главной программы готов. Требуется его откомпилировать с использованием библиотеки **dl**:

```
[gun]$ gcc main1.c -o main1 -ldl
```

Получим программный файл main1, который можно тестировать.

### 4.5.5. Инициализация и деинициализация динамических библиотек

Представим себе ситуацию, когда функции библиотеки для работы требуют правильно инициализированные переменные, например если для работы функции нужен буфер или массив. Для таких случаев в библиотеках можно задавать инициализирующую и деинициализирующую функции:

```
void _init(); //инициализация
void _fini(); //деинициализация
```

Поясним их применение на примере. Введем в нашей библиотеке lib.c глобальную переменную **test** и возвращающую ее функцию:

```
char *test; char *ret_test(){ return test; };
```

Перепишем основную программу main2.c (см. [6]):

```
#include <stdio.h>
#include <dlfcn.h>
int main(){
void *ext_library; double value=0;
char * (*ret_test)();
ext_library = dlopen ("libtest.so",RTLD_LAZY);
if (!ext_library){
fprintf(stderr,"dlopen() error: %s\n", dlerror());
return 1; };
ret_test = dlsym (ext_library,"ret_test");
printf("Return of ret_test:\"%s\" [%p]\n",
(*ret_test)(), (*ret_test)());
dlclose(ext_library); };
```

После компиляции этого примера мы получим результат:

```
[gun]$ gcc -c lib.c -fPIC
[gun]$ gcc -shared lib.o -o libtest.so
[gun]$ gcc -o main2 main2.c -ldl
[gun]$ ./main Return of ret_test: "(null)" [(nil)]
```

Как видим, переменная **test** оказалась равной **NULL**, а хотелось бы иметь в ней значение. Для этого посмотрим, как работают функции **_init()** и **_fini()**. Создадим вторую библиотеку lib1.c (см. [6]):

```
#include <stdlib.h>
char *test; char *ret_test(){ return test; };
void _init(){
test=(char *)malloc(5);
if (test!=NULL){
*(test+0)='g'; *(test+1)='u'; *(test+2)='n';
*(test+3)='!'; *(test+4)=0;
 };
printf("_init() executed...\n"); };
void _fini(){
if (test!=NULL) free(test);
printf("_fini() executed...\n"); };
```

Теперь пробуем откомпилировать:

```
[gun]$ gcc -c lib1.c -fPIC
[gun]$ gcc -shared lib1.o -o libtest.so
lib1.o: In function `_init': lib1.o(.text+0x24):
multiple definition of `_init'
/usr/lib/crti.o(.init+0x0): first defined here
lib1.o:
In function `_fini': lib1.o(.text+0xc0): multiple
definition of `_fini' /usr/lib/crti.o(.fini+0x0):
first defined here collect2: ld returned 1 exit
status
```

Программа не может скомпоноваться. Чтобы избавиться от мешающей библиотеки надо использовать ключ компилятора **-nostdlib**. Попробуем пересобрать библиотеку:

```
[gun]$ gcc -shared -nostdlib lib1.o -o libtest.so
```

Теперь повторно запускаем **main**:

```
[gun]$ ./main2
_init() executed...
Return of ret_test: "gun!" [0x8049c20]
_fini() executed...
```

Как видим, глобальная (для библиотеки) переменная **test** была проинициализирована при загрузке, ее значение выведено библиотечной функцией, а при выгрузке библиотеки динамически выделенная для переменной память была освобождена.

## Вопросы для самопроверки

1. В чем состоит необходимость организации библиотек объектов?
2. Дайте определение библиотеки объектных файлов.
3. Дайте определение статической и динамической библиотеки.
4. Опишите команду создания и редактирования статических библиотек.
5. Перечислите варианты подключения статических библиотек.
6. Каковы особенности разделяемых библиотек?
7. Опишите процесс создания разделяемых библиотек.
8. Что такое адресно-независимый код?
9. Каковы особенности применения разделяемых библиотек?

10. Опишите команды **objdump**, **readelf**, **nm**.
11. Что такое динамически загружаемые библиотеки?
12. Перечислите функции интерфейса **dlopen**.
13. Каковы особенности применения функции **dlopen()**?
14. Опишите флаги, применяемые в функции **dlopen()**.
15. Каковы особенности применения функции **dlsym()**?

## Упражнения

Выполните лабораторную работу № 2 из лабораторного практикума.

# Глава 5. Многозадачное программирование в Linux

## 5.1. Основные системные вызовы для реализации многозадачности

Основными элементами для реализации многозадачности являются системные вызовы **`fork()`**, **`exit()`**, **`wait()`** и **`execve()`**. Каждый из них будет детально рассмотрен далее, здесь же ознакомимся с кратким описанием этих четырех вызовов и их использования в связке друг с другом.

Системный вызов **`fork()`** позволяет одному процессу, родителю, создавать новый, дочерний процесс. Оба этих процесса являются (почти) идентичными: потомок получает копии родительского стека, данных, кучи, копии родительских сегментов стека и текста. Термин «fork» («вилка», «разветвление») стали применять потому, что родительский процесс как бы делится на две копии самого себя.

Библиотечная функция **`exit(status)`** завершает процесс, делая все его ресурсы (память, дескрипторы открытых файлов и т. д.) доступными для последующего перераспределения ядром. Аргумент **`status`** – целое число, которое определяет код завершения процесса. Родительский процесс может извлечь этот код с помощью системного вызова **`wait()`**.

Системный вызов **`wait(&status)`** имеет два назначения. Во-первых, если работа потомка текущего процесса еще не была завершена путем вызова **`exit()`**, функция **`wait()`** приостанавливает выполнение родителя, пока не будет завершен один из его потомков. Во-вторых, код завершения потомка возвращается через аргумент функции **`wait()`**.

Системный вызов **execve(pathname, argv, envp)** загружает в память процесса новую программу (расположенную в **pathname**, с аргументами **argv** и списком переменных среды **envp**). Текст существующей программы сбрасывается, а для новой программы заново создаются сегменты со стеком, данными и кучей. Эту операцию часто называют *выполнением* новой программы.

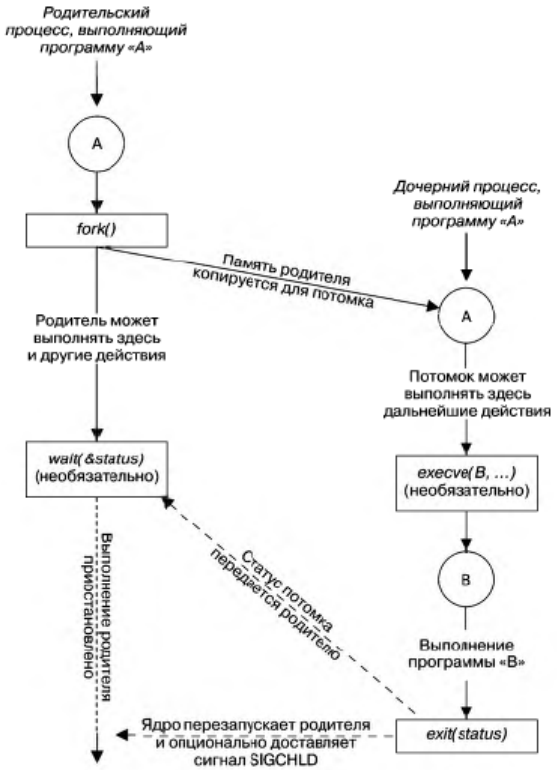



*Рис. 5.1.* Обзор вызовов **fork()**, **exit()**, **wait()** и **execve()**

На рис. 5.1 показано, как вызовы **fork()**, **exit()**, **wait()** и **execve()** обычно используются вместе. Применение вызова **execve()**, показанное на этой диаграмме, не является обязательным. Иногда имеет смысл позволить потомку продолжить выполнение про-

граммы родителя. В любом случае выполнение дочернего процесса в конечном счете завершается вызовом **exit()** (или передачей сигнала) и возвращением кода завершения, доступным родителю через функцию **wait()**. Вызов **wait()** тоже необязателен. Родитель может просто игнорировать своего потомка и продолжать работу. Однако использование функции **wait()** обычно является желательным и выполняется внутри обработчика сигнала **SIGCHLD**, который генерируется ядром для родителя, когда один из его дочерних процессов завершается (по умолчанию сигнал **SIGCHLD** игнорируется, поэтому в диаграмме сказано, что его доставка является опциональной).

## 5.2. Идентификаторы процессов в Linux

Стандартная библиотека C (libc, реализованная в Linux в glibc) использует возможности многозадачности Unix System V (далее SysV). В многозадачной ОС любой процесс обладает уникальным идентификатором процесса (PID, Process Identifier – идентификатор процесса), представляющим собой целое положительное число, уникальность которого гарантируется системой.

В libc тип **pid_t** определен как целое, способное вместить в себе PID. Рассмотрим функцию, которая сообщает идентификатор процесса, содержащего программу (она определена вместе с **pid_t** в файлах unistd.h и sys/types.h):

```
pid_t getpid (void);
```

Напишем программу, выводящую в стандартный вывод свой идентификатор:

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
int main() {
pid_t pid; pid = getpid();
printf("pid, присвоенный процессу - %d\n", pid);
return 0; }
```

Эта программа выведет свой PID, и при последующих запусках это число будет постоянно увеличиваться, потому что в перерыве между запусками может быть создан другой процесс. Это можно выяснить, выполняя утилиту **ps** между запусками примера.

## 5.3. Порождение процессов

Системный вызов **fork()** создает новый процесс – *потомок*, который является почти полной копией вызывающего процесса, *родителя*:

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

В родительском процессе функция возвращает идентификатор потомка при успешном завершении или –1 при ошибке, в успешно созданном потомке всегда возвращает 0.

Оба процесса выполняют один и тот же программный код, но обладают разными копиями сегментов стека, данных и кучи. Сегменты потомка вначале полностью дублируют соответствующие части памяти своего родителя. Но после завершения вызова **fork()** каждый из процессов может самостоятельно изменять переменные в своих сегментах, не влияя на другой процесс.

Ключевым моментом в понимании вызова **fork()** является тот факт, что после завершения его работы мы получаем два процесса, каждый из которых продолжает выполнение с момента возврата из этого вызова. В случае одного процессорного ядра решение, какой процесс должен выполняться (родительский или дочерний), принимается диспетчером задач.

Распознавать процессы внутри кода программы можно с помощью значения, возвращенного функцией **fork()**. В случае с родителем это значение равно идентификатору только что созданного потомка. Это полезно, поскольку родитель может создать несколько дочерних процессов, которые ему придется отслеживать (с помощью вызова **wait()** или одной из его разновидностей). В случае с потомком возвращается 0. При необходимости дочерний процесс может получить свой собственный идентификатор или идентификатор своего родителя, используя функции **getpid()** и соответственно **getppid()**.

Если новый процесс не удается создать, вызов **fork()** возвращает –1. Одной из причин этого может быть превышение пользователем или всей системой в целом ограничения на количество создаваемых процессов.

Следует понимать, что после вызова **fork()** невозможно сказать, какой из двух процессов первым получит от планировщика ресурсы процессора. В плохо написанных программах такая неопределенность может привести к ошибке, известной под названием *«состояние гонки»*.

Оба процесса содержат коды, как родительские, так и дочерние, однако оба они должны выполнить только свой набор кодов. Чтобы прояснить это, взглянем на алгоритм:

```
РАЗВЕТВИТЬ
ЕСЛИ ТЫ ДОЧЕРНИЙ ПРОЦЕСС ВЫПОЛНИТЬ (...)
ЕСЛИ ТЫ РОДИТЕЛЬСКИЙ ПРОЦЕСС ВЫПОЛНИТЬ (...)
```

который представляет собой код, написанный на некоем метаязыке.

На языке С получим:

```
int main() {
  pid_t pid;
  pid = fork();
  if (pid == 0)
  { КОД ДОЧЕРНЕГО ПРОЦЕССА }
  КОД РОДИТЕЛЬСКОГО ПРОЦЕССА
}
```

Когда выполняется вызов **fork()**, потомок получает копии всех файловых дескрипторов родителя. Копии в том смысле, что соответствующие дескрипторы в родительском и дочернем процессах указывают на один и тот же открытый файл. Дескриптор открытого файла содержит его текущее смещение (образовавшееся во время редактирования с помощью функций **read()**, **write()** и **lseek()**) и флаги состояния (установленные вызовом **open()** и измененные операцией **fcntl() F_SETFL**). Как следствие, эти атрибуты открытого файла являются общими для родителя и потомка. Если, например, дочерний процесс обновляет смещение в файле, то это изменение доступно его родителю посредством соответствующего дескриптора.

Linux 3.2.0 дает возможность создавать новые процессы с помощью системного вызова **clone**. Это более универсальный вариант функции **fork**, позволяющий вызывающему процессу определить, что будет совместно использоваться дочерним и родительским процессами.

## 5.4. Методы синхронизации процессов

Часто родительскому процессу необходимо синхронизироваться с дочерними процессами, чтобы выполнять операции в нужное время. Основной способ синхронизации процессов – системные вызовы **wait** и **waitpid**.

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

Системный вызов **wait()** ждет, когда один из потомков вызывающего процесса прекратит работу и возвращает код завершения этого дочернего процесса через буфер, на который указывает аргумент **status**. Возвращает идентификатор завершенного процесса или –1 при ошибке. При этом:

1) если ни один из потомков вызывающего процесса еще не завершился, вызов блокируется, пока этого не произойдет. Если на момент вызова какой-либо потомок уже прекратил работу, **wait()** сразу же возвращает значение;

2) если параметр **status** не равен **NULL**, он указывает на целое число, описывающее подробности завершения потомка;

3) вызов **wait()** возвращает идентификатор завершившегося дочернего процесса.

В случае ошибки **wait()** возвращает –1. К ошибке может привести, например, отсутствие у вызывающего процесса потомков; в этом случае **errno** присваивается значение **ECHILD**. Это означает, что мы можем ждать завершения всех потомков вызывающего процесса в следующем цикле:

```
while ((childPid = wait(NULL)) != -1)
continue;
if (errno != ECHILD) / * Непредвиденная ошибка... * /
printf ( "Error while waiting…") ;
```

Системный вызов **waitpid()** снимает ряд ограничений, присущих вызову **wait()**.

- Если родительский процесс создал несколько потомков, **wait()** не позволяет ожидать завершения конкретного из них; мы можем отслеживать завершение работы только следующего дочернего процесса.

• Если ни один из потомков еще не был завершен, вызов **wait()** всегда блокируется. Иногда имеет смысл организовать неблокирующее ожидание, чтобы, если все потомки все еще работают, иметь возможность немедленно узнать об этом факте.

• С помощью **wait()** можно отслеживать только потомки, которые завершились. Если потомок был остановлен (по сигналам **SIGSTOP** или **SIGTTIN**) или возобновил свою работу (по сигналу **SIGCONT**), мы об этом не узнаем.

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

Возвращает идентификатор потомка: 0 (см. далее) или –1, если возникла ошибка.

Возвращаемое значение и аргумент **status** у вызова **waitpid()** служат тем же целям, что и у **wait()**. Аргумент **pid** позволяет выбрать потомки, которые будут отслеживаться. Делается это следующим образом:

• если **pid** больше 0, ожидаем потомка, чей *идентификатор* равен **pid**;

• если **pid** равен 0, ожидаем потомка, входящего *в ту же группу процессов, что и родитель*;

• если **pid** меньше –1, ожидаем любого потомка, идентификатор *группы процессов* которого равен значению **pid** по модулю;

• если **pid** равен –1, ожидаем *любого* потомка. Вызов **wait(&status)** эквивалентен вызову **waitpid(-1, &status, 0)**.

Аргумент **options** представляет собой битовую маску, которая может быть пустой (равной 0) или содержать какое-то количество следующих флагов (каждый из которых входит в стандарт SUSv3), скомбинированных с помощью побитового ИЛИ:

• **WUNTRACED** – позволяет узнать не только о завершенных дочерних процессах, но и о потомках, *остановленных* с помощью сигнала;

• **WCONTINUED** (начиная с Linux 2.6.10) – позволяет узнать о потомках, работа которых была возобновлена после остановки в результате получения сигнала **SIGCONT**;

• **WNOHANG** – если ни один из потомков, указанных в **pid**, все еще не изменил свое состояние, вызов немедленно возвращается, не пере-

ходя к блокированию (то есть «опрашивает» потомков). В этом случае возвращаемое значение **waitpid()** равно 0.

Если у родителя не оказывается дочерних процессов, соответствующих значению **pid**, **waitpid()** возвращает ошибку **ECHILD**.

Значение **status**, возвращаемое вызовами **wait()** и **waitpid()**, позволяет различать следующие события, касающиеся потомков;

- потомок завершил работу с помощью вызова **_exit()** (или **exit ()**), вернув целочисленный *код выхода*;
- потомок был завершен путем доставки необработанного сигнала;
- потомок был остановлен сигналом, а вызов **waitpid()** был произведен с использованием флага **WUNTRACED**;
- потомок возобновил работу после сигнала **SIGCONT**, а вызов **waitpid()** был выполнен с использованием флага **WCONTINUED**.

Чтобы охватить все случаи, приведенные выше, мы будем обращаться к термину «статус ожидания». Первые два пункта можно назвать *кодом завершения*. И хотя значение **status** является целым числом, только последние два байта из него действительно используются. Способ их заполнения зависит от того, какое из перечисленных выше событий произошло с потомком.

Заголовочный файл <sys/wait.h> определяет стандартный набор макросов, с помощью которых можно анализировать код завершения. Только один из перечисленных в списке ниже вернет **true**, если применить его к значению **status**, возвращаемому вызовами **wait()** и **waitpid()**. Для более глубокого анализа предоставляются дополнительные макросы:

- **WIFEXITED(status)** – возвращает **true**, если дочерний процесс завершился штатно. В этом случае макрос **WEXITSTATUS(status)** возвращает код завершения дочернего процесса (родителю доступен только младший байт кода завершения).
- **WIFSIGNALED(status)** – возвращает **true**, если дочерний процесс был завершен с помощью сигнала. В этом случае макрос **WTERMSIG(status)** возвращает номер сигнала, приведшего к завершению процесса, а макрос **WCOREDUMP(status)** возвращает **true**, если потомок сгенерировал файл с дампом памяти. **WCOREDUMP(status)** не входит в стандарт SUSv3, но доступен в большинстве реализаций UNIX.

• **WIFSTOPPED(status)** – возвращает **true**, если дочерний процесс был остановлен по сигналу. В этом случае макрос **WSTOPSIG(status)** возвращает номер сигнала, остановившего процесс.

• **WIFCONTINUED (status)** – возвращает **true**, если дочерний процесс возобновил свою работу, получив сигнал **SIGCONT**. Он доступен в системе Linux начиная с версии 2.6.10.

Обратите внимание, что, хотя вышеприведенные макросы тоже используют для своих аргументов имя **status**, они ожидают получить обычное целое число, а не указатель на него, как в случае с вызовами **wait()** и **waitpid()**.

Если дочерний процесс заканчивает работу раньше родительского, то родительский процесс не сможет получить его код завершения, когда это потребуется. Ядро сохраняет некоторый объем информации о каждом завершившемся процессе, чтобы она была доступна, когда родительский процесс вызовет функцию **wait()** или **waitpid()**. В простейшем случае эта информация состоит из идентификатора процесса и кода завершения. Ядро может освободить всю память, занимаемую процессом, и закрыть его открытые файлы. В терминологии UNIX процесс, который завершился, но при этом его родительский процесс не уловил этого момента, называют *зомби*. Команда **ps** выводит в поле состояния процесса-зомби символ Z. Если написать долго работающую программу, которая порождает множество дочерних процессов, они будут превращаться в зомби, если программа не станет дожидаться получения от них кодов завершения.

## 5.5. Завершение процесса

Обычно процесс можно завершить двумя способами. Во-первых, это *аварийное* завершение, вызванное передачей сигнала, чьим действием по умолчанию является остановка работы процесса (со сбросом дампа памяти или без него). Варианты аварийного завершения процесса рассмотрим позже.

Возможно также *нормальное* завершение с помощью системного вызова **_exit()**.

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

Аргумент **status**, передаваемый вызову **_exit()**, определяет код завершения процесса, который доступен его родителю посредством вызова **wait()**. Этот аргумент имеет тип **int**, однако родительскому процессу доступны только последние 8 бит. Код 0 принято считать признаком успешного завершения, а любые другие значения сигнализируют о том, что процесс окончил работу с проблемами. Четких правил по интерпретации ненулевых значений не существует; разные приложения следуют своим собственным соглашениям, которые должны быть описаны в их документации. Стандарт SUSv3 определяет две константы – **EXIT_SUCCESS(0)** и **EXIT_FAILURE(1)**.

Процесс всегда завершается успешно, если для этого применяется вызов **_exit()** (т. е. системная функция **_exit()** никогда не возвращает значения).

Программы обычно не вызывают **_exit()** напрямую, используя вместо этого библиотечную функцию **exit()**, которая выполняет различные предварительные действия.

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

Функция **exit()** выполняет следующие приготовления перед вызовом **_exit()**.

1. Вызываются обработчики выхода (функции, зарегистрированные с помощью вызовов **atexit()** и **on_exit()**, здесь не рассматриваются), в порядке, обратном их регистрации.

2. Сбрасываются буферы потоков **stdio**.

3. Выполняется системный вызов **_exit()** со значением, переданным в аргументе **status**.

В отличие от вызова **_exit()**, характерного для UNIX-систем, функция **exit()** является частью стандартной библиотеки C; это означает, что она доступна в любой реализации данного языка.

Другим способом завершения работы процесса является возвращение из функции **main()**, явное или неявное (когда достигается конец функции). Явный возврат значения **(return n)** обычно является эквивалентом вызова **exit(n)**, поскольку среда выполнения, которая вызывает функцию **main()**, использует возвращаемое из нее значение для вызова **exit()**.

Возвращение из функции без указывания значения или просто завершение главной функции тоже приводит к тому, что программа, вы-

звавшая функцию **main()**, выполняет вызов **exit()**, однако результат при этом зависит от поддерживаемого стандарта языка С и использованных параметров компиляции.

- В стандарте С89 поведение при таких обстоятельствах не определено; программа может завершиться с произвольным статусом. Так, например, происходит на платформе Linux в сочетании с компилятором **gcc**: код завершения программы берется из какого-то случайного значения в стеке или определенном регистре процессора. Следует избегать завершения программ таким образом.
- Стандарт С99 требует, чтобы окончание функции **main()** было эквивалентно вызову **exit(0)**. Для того чтобы программа в системе Linux вела себя именно так, нужно скомпилировать ее с использованием параметра **gcc -std=c99**.

## 5.6. Функции и программы в порожденных процессах

Чтобы дочерний процесс выполнял действия с неким специфическим набором данных, в качестве дочернего процесса можно вызвать функцию из адресного пространства родителя:

```
pid = fork();
if (pid == 0) {
// если дочерний процесс, то вызовем функцию
res=process(arg);
// выход из дочернего процесса
exit(res);
}
```

Часто в качестве дочернего процесса необходимо запускать другую программу. Для этого применяется 7 функций семейства **exec**:

```
#include <unistd.h>
int execl(const char *pathname, const char
*arg0,./* (char *)0 */);
int execv(const char *pathname, char *const
argv[]);
int execle(const char *pathname, const char *arg0, ...
/* (char *)0, char *const envp[] */ );
```

```
int execve(const char *pathname, char *const
argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0,
... /* (char *)0 */ );
int execvp(const char *filename, char *const
argv[]);
int fexecve(int fd, char *const argv[], char *const
envp[]);
```

Все семь функций возвращают –1 в случае ошибки, не возвращают управление в случае успеха.

Одно из отличий между этими функциями заключается в том, что первые четыре принимают полный путь к файлу, следующие две – только имя файла и последняя – дескриптор файла. Аргумент **filename** интерпретируется следующим образом:

- если аргумент **filename** содержит символ слеша, он интерпретируется как полный путь к файлу;
- иначе производится поиск выполняемого файла в каталогах, перечисленных в переменной окружения **PATH**.

Если функция **execlp** или **execvp** находит выполняемый файл, используя один из префиксов пути, но этот файл не является двоичным выполняемым файлом, сгенерированным редактором связей, то функция предположит, что найденный файл является сценарием командной оболочки, и попытается вызывать **/bin/sh** с именем файла в качестве аргумента.

Функция **fexecve** вообще перекладывает задачу поиска выполняемого файла на вызывающую программу. Используя файловый дескриптор, вызывающая программа может проверить, действительно ли файл является тем, который требуется выполнить, и запустить его.

И последнее различие – передача списка переменных окружения новой программе. Три функции, имена которых оканчиваются на **e** (**execle**, **execve** и **fexecve**), позволяют передать массив указателей на новые строки окружения. Остальные четыре функции для передачи копии среды окружения новой программе используют переменную **environ** вызывающего процесса.

Аргументы всех семи функций семейства **exec** достаточно сложно запомнить. Но буквы в именах функций немного помогают в этом. Буква **p** означает, что функция принимает аргумент **filename** и ис-

пользует переменную окружения **PATH**, чтобы найти выполняемый файл. Буква **l** означает, что функция принимает список аргументов, а буква **v** – что она принимает массив (вектор) **argv[]**. Наконец, буква **e** означает, что функция принимает массив **envp[]** вместо текущего окружения.

Аргументы **arg**,... функций **execl**, **execlp**, **execle** составляют список указателей на символьные строки, содержащие параметры, передаваемые программе. По соглашениям первый элемент этого списка должен содержать имя программного файла. Список параметров должен заканчиваться пустым указателем – **NULL** или **(char *)0**.

В качестве примера на сайте [6], в разделе «Примеры программ», «Linuxprog», имеется программа spaces.c, порождающая столько дочерних процессов, сколько аргументов командной строки было ею получено, и запускающая в каждом дочернем процессе программу file.c с очередным аргументом командной строки родителя в качестве входного параматра, ждущая завершения всех дочерних процессов и получающая коды их завершения.

## 5.7. Управление приоритетами процессов

Традиционно системами UNIX поддерживается лишь очень грубая настройка приоритетов процессов для планирования. Политика планирования и поддержки приоритетов определяется ядром. Процесс может понизить свой приоритет, изменив так называемый коэффициент уступчивости (т. е. изменив значение коэффициента уступчивости («nice»), процесс может стать более «уступчивым» и уменьшить выделяемую ему долю процессорного времени). Только привилегированные процессы могут увеличивать свой приоритет.

Согласно стандарту Single UNIX Specification, значения коэффициента уступчивости изменяются в диапазоне от **0** до **(2*NZERO)–1**, хотя некоторые реализации поддерживают диапазон от **0** до **2*NZERO**. Более низкие значения коэффициента уступчивости соответствуют более высокому приоритету. Константа **NZERO** определяет значение по умолчанию для коэффициента уступчивости в данной системе.

Функция **getpriority** позволяет получить значение коэффициента уступчивости процесса. Но помимо этого **getpriority** может

также вернуть значение коэффициента уступчивости для группы родственных процессов.

```
#include <sys/resource.h>
int getpriority(int which, id_t who);
```

Функция возвращает значение коэффициента уступчивости между **–NZERO** и **NZERO–1** в случае успеха, –1 в случае ошибки. Параметр **which** определяет, должен ли системный вызов работать с приоритетом процесса, группы процессов или пользователя, его возможные значения – **PRIO_PROCESS**, **PRIO_PGRP** или **PRIO_USER** соответственно. Параметр **who** задает идентификатор процесса, группы процессов или пользователя – в зависимости от значения параметра **which**.

Если в аргументе **who** передать 0, операция выполнится для текущего процесса, группы процессов или пользователя (в зависимости от значения аргумента **which**).

Если аргумент **which** соответствует более чем одному процессу, возвращается наивысший приоритет (наименьшее значение) из всех выбранных процессов.

Для изменения приоритета процесса, группы процессов или всех процессов, принадлежащих определенному пользователю, можно использовать функцию **setpriority**.

```
#include <sys/resource.h>
int setpriority(int which, id_t who, int value);
```

Функция возвращает 0 в случае успеха, –1 в случае ошибки.

Назначение аргументов **which** и **who** в точности соответствует назначению одноименных аргументов функции **getpriority**. К значению **value** добавляется константа **NZERO**, и результат становится новым значением коэффициента уступчивости.

## 5.8. Ненормальное завершение процесса. Сигналы

Как упоминалось ранее (п. 5.4), возможно *аварийное* завершение процесса, вызванное передачей сигнала, чьим действием по умолчанию является остановка работы процесса.

Каждый сигнал имеет собственное имя. Имена всех сигналов начинаются с последовательности SIG. Например, **SIGABRT** – это сигнал прерывания, который генерируется, когда процесс вызывает функцию **abort**. Linux 3.2.0 содержит 31 сигнал, имена которых определены как константы с положительными числовыми значениями (номерами сигналов) в заголовочном файле <signal.h>.

Сигналы могут порождаться различными условиями.

Сигналы, генерируемые терминалом, возникают, когда пользователь вводит определенные символы. Нажатие клавиш **Control-C** порождает сигнал прерывания (**SIGINT**). Таким способом можно прервать выполнение программы, вышедшей из-под контроля.

Аппаратные ошибки – деление на 0, ошибка доступа к памяти и прочее – также приводят к генерации сигналов. Эти ошибки обычно обнаруживаются аппаратным обеспечением, которое извещает ядро об их появлении. После этого ядро генерирует соответствующий сигнал и передает его процессу, который выполнялся в момент появления ошибки. Например, сигнал **SIGSEGV** посылается процессу при попытке обратиться к неверному адресу в памяти.

Функция **kill()** позволяет процессу передать любой сигнал другому процессу или группе процессов. Естественно, здесь существуют свои ограничения: необходимо быть владельцем процесса, которому посылается сигнал, или обладать привилегиями суперпользователя.

Сигналы являют собой классический пример асинхронных событий. Сигнал может быть передан процессу в любой момент. Чтобы выяснить причину, породившую сигнал, процесс может запросить ядро произвести одно из трех действий, связаннымх с сигналом.

1. Игнорировать сигнал. Это действие возможно для большинства сигналов, но два сигнала, **SIGKILL** и **SIGSTOP**, нельзя игнорировать. Причина, почему эти два сигнала не могут быть проигнорированы, заключается в том, что ядру и суперпользователю необходима возможность завершить или остановить любой процесс. Кроме того, если проигнорировать некоторые из сигналов, возникающих в результате аппаратных ошибок (таких, как деление на 0 или попытка обращения к несуществующей памяти), поведение процесса может стать непредсказуемым.

2. Перехватить сигнал. Для этого нужно сообщить ядру адрес функции, которая будет обрабатывать сигнал. В этой функции можно предусмотреть действия по обработке условия, породившего сигнал. Если пойман сигнал **SIGCHLD**, который означает завершение дочернего процесса,

то функция, перехватившая сигнал, может вызвать функцию **waitpid()**, чтобы получить идентификатор дочернего процесса и код его завершения. Еще один пример: если процесс создает временные файлы, имеет смысл написать функцию обработки сигнала **SIGTERM**, которая будет удалять временные файлы. Сигналы **SIGKILL** и **SIGSTOP** нельзя перехватить.

3. Применить действие по умолчанию. Каждому сигналу поставлено в соответствие некоторое действие по умолчанию (перечислены в таблице). Заметьте, что для большинства сигналов действие по умолчанию заключается в завершении процесса.

В таблице перечислены имена некоторых сигналов и указано, каково действие по умолчанию для каждого сигнала. Если в колонке «Действие по умолчанию» указано «завершить + core», это означает, что образ памяти процесса сохраняется в файле core в текущем рабочем каталоге процесса. Большинство отладчиков могут использовать этот файл для выяснения причин, породивших преждевременное завершение процесса.

**Некоторые сигналы и действия по умолчанию**

| Имя | Описание | Действие по умолчанию |
|---|---|---|
| **SIGABRT** | Аварийное завершение (abort) | Завершить + core |
| **SIGBUS** | Аппаратная ошибка | Завершить + core |
| **SIGCHLD** | Изменение состояния дочернего процесса | Игнорировать |
| **SIGCONT** | Возобновить работу приостановленного процесса | Продолжить/ игнорировать |
| **SIGFPE** | Арифметическая ошибка | Завершить + core |
| **SIGHUP** | Обрыв связи с терминалом | Завершить |
| **SIGILL** | Недопустимая инструкция | Завершить + core |
| **SIGINT** | С терминала введен символ прерывания | Завершить |
| **SIGKILL** | Завершение | Завершить |
| **SIGSEGV** | Ошибка доступа к памяти | Завершить + core |
| **SIGSTOP** | Приостановить процесс | Остановить процесс |
| **SIGTERM** | Завершение | Завершить |
| **SIGUSR1** | Определяемый пользователем сигнал | Завершить |
| **SIGUSR2** | Определяемый пользователем сигнал | Завершить |

Опишем подробнее несколько сигналов, используемых для управления процессами.

**SIGABRT.** Генерируется вызовом функции **abort**. Процесс завершается аварийно.

**SIGCHLD.** Когда процесс завершается или останавливается, родительскому процессу посылается сигнал **SIGCHLD**. По умолчанию этот сигнал игнорируется, но родительский процесс может перехватить его, если желает получать извещения об изменении состояния дочерних процессов. Функция, перехватывающая этот сигнал, обычно вызывает одну из функций семейства **wait**, чтобы получить идентификатор дочернего процесса и код завершения.

**SIGKILL.** Один из двух сигналов, которые нельзя перехватить или игнорировать в приложении. Дает возможность системному администратору уничтожить любой процесс.

**SIGTERM.** Сигнал завершения процесса. Так как его можно перехватить, обработка сигнала **SIGTERM** дает программам возможность корректно завершить работу, освободив занятые ресурсы (в противоположность сигналу **SIGKILL**, который нельзя перехватить или игнорировать).

Простейшим интерфейсом к сигналам UNIX служит функция **signal**:

```
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);
```

Она возвращает предыдущую функцию обработки сигнала в случае успеха, **SIG_ERR** в случае ошибки.

Аргумент **signo** – это имя сигнала из табл. 5.1. В качестве аргумента **func** можно передать константу **SIG_IGN**, или константу **SIG_DFL**, или адрес функции, которая будет вызвана при получении сигнала. Константа **SIG_IGN** сообщает системе, что сигнал должен игнорироваться. (Не забывайте, что два сигнала, **SIGKILL** и **SIGSTOP**, не могут игнорироваться.) Если указана константа **SIG_DFL**, с сигналом связывается действие по умолчанию (последняя колонка в табл. 5.1). Если указан адрес функции, она будет вызываться при получении сигнала, то есть будет «перехватывать» сигнал. Такие функции называются *обработчиками* или *перехватчиками* сигналов.

Функции – обработчику сигнала передается единственный аргумент (целое число – номер сигнала), и она ничего не возвращает. Когда функция **signal** вызывается, чтобы установить обработчик сигнала, второй аргумент должен быть указателем на функцию. Возвращаемое значение функции **signal** – указатель на предыдущий обработчик сигнала.

Функция **kill** посылает сигнал процессу или группе процессов:

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

возвращает 0 в случае успеха, –1 в случае ошибки.

Интерпретация аргумента **pid** функции **kill()** производится в соответствии со следующими правилами:

- если **pid** > 0, то он задает идентификатор процесса, которому посылается сигнал;
- если **pid** == 0, то сигнал посылается всем процессам той группы, к которой принадлежит текущий процесс;
- если **pid** < 0, то сигнал посылается всем процессам с идентификатором группы процессов, равным абсолютному значению **pid**, которым данный процесс имеет право посылать сигналы. Опять же в понятие «все процессы» не входят системные процессы, определяемые реализацией;
- если **pid** == –1, то сигнал посылается всем процессам в системе, которым данный процесс имеет право посылать сигналы.

Стандарт POSIX определяет сигнал с номером 0 как пустой сигнал. Если аргумент **signo** имеет значение 0, функция **kill()** выполнит обычную проверку на наличие ошибок, но сам сигнал не пошлет. Это часто используется, чтобы определить, существует ли еще некоторый процесс. Если несуществующему процессу послать пустой сигнал, функция **kill()** вернет –1 и код ошибки **ESRCH** в переменной **errno**. К моменту, когда функция **kill()** вернет управление в вызывающую программу, проверяемый процесс уже может завершиться, что сильно ограничивает область применения такого приема.

Если в результате вызова функции **kill()** генерируется сигнал и при этом сигнал не заблокирован, тогда сигнал с номером **signo** будет доставлен процессу еще до того, как функция **kill()** вернет управление.

## Вопросы для самопроверки

1. Что такое **pid** и как его получить?
2. Что делает и что возвращает функция **fork()**?
3. Чем отличается дочерний процесс от родительского сразу после вызова функции **fork()**?
4. Что делает и что возвращает функция **wait()**?
5. Чем функция **waitpid()** отличается от **wait()**?
6. Каковы параметры и возвращаемое значение функции **waitpid()**?
7. Как получить код завершения дочернего процесса?
8. Перечислите макросы для анализа кода завершения дочернего процесса.
9. Какой процесс называют «зомби»?
10. Перечислите методы нормального завершения процесса.
11. Чем отличаются функции **exit()** и **_exit()**?
12. Как в дочернем процессе запустить другую программу?
13. Как определить и изменить приоритет процесса?
14. Какие сигналы можно использовать для управления процессом?
15. Перечислите допустимые параметры функции **kill()**.
16. Каковы особенности сигнала **SIGCHLD**? Как его перехватить?

## Упражнения

Выполните лабораторную работу № 3 из лабораторного практикума.

# Глава 6. Linux IPC

## 6.1. Совместное использование информации процессами

В модели программирования UNIX в системе могут одновременно выполняться несколько процессов, каждому из которых выделяется собственное адресное пространство. Это иллюстрирует рис. 6.1 [2].

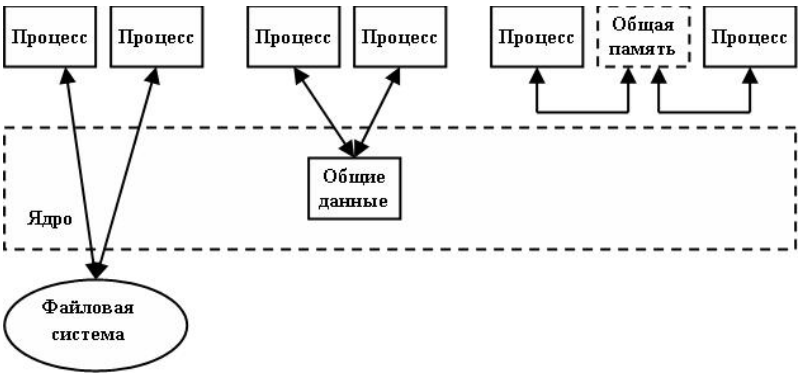



*Рис. 6.1*. Совместное использование информации процессами

Два процесса в левой части совместно используют информацию, хранящуюся в одном из объектов файловой системы. Для доступа к этим данным каждый процесс должен обратиться к ядру (используя функции **`read`**, **`write`**, **`seek`** и аналогичные).

Два процесса в середине рисунка совместно используют информацию, хранящуюся в ядре. Примерами являются канал, очередь сообщений или семафор System V. Для доступа к совместно используемой информации в этом случае будут использоваться системные вызовы.

Два процесса в правой части используют общую область памяти, к которой может обращаться каждый из процессов. После того как будет получен доступ к этой области памяти, процессы смогут обращаться к данным вообще без помощи ядра. Процессам, использующим общую память, также требуется синхронизация.

Такое взаимодействие процессов часто называют аббревиатурой IPC (InterProcess Communication, межпроцессное взаимодействие).

Исторически первым средством взаимодействия процессов в ОС UNIX являлись неименованные (pipes) и именованные (FIFO) программные каналы. Именованный канал служит для общения и синхронизации произвольных процессов, знающих имя данного канала и имеющих соответствующие права доступа. Неименованным каналом могут пользоваться только создавший его процесс и его потомки.

Существует два подхода к организации IPC: System V IPC (далее Sys V) и POSIX IPC. Первый является «родным» для UNIX и с его помощью реализовано большое количество существующих приложений. Второй призван обеспечить переносимость программного обеспечения.

В SysV IPC входят:

- очереди сообщений System V;
- семафоры System V;
- общая память System V.

У них много общего: схожи функции, с помощью которых организуется доступ к объектам; также схожи формы хранения информации в ядре. Информация о функциях сведена в табл. 6.1.

Таблица 6.1

**Функции System V IPC**

| | Очереди сообщений | Семафоры | Общая память |
|---|---|---|---|
| Заголовочный файл | **<sys/ msg.h>** | **<sys/ sem.h>** | **<sys/ shm.h>** |
| Создание или открытие | **msgget** | **semget** | **shmget** |
| Операции управления | **msgctl** | **semctl** | **shmctl** |
| Операции IPC | **msgsnd, msgrcv** | **semop** | **shmat, shmdt** |

Из имеющихся типов IPC следующие три могут быть отнесены к POSIX IPC:

- очереди сообщений POSIX IPC;
- семафоры POSIX IPC;
- общая память POSIX IPC.

Информация о функциях сведена в табл. 6.2.

Таблица 6.2

**Функции POSIX IPC**

| | Очереди сообщений | Семафоры | Общая память |
|---|---|---|---|
| Заголовочный файл | `<mqueue.h>` | `<semaphore.h>` | `<sys/ mman.h>` |
| Создание или открытие | `mq_open` | `sem_open` | `shm_open` |
| Операции управления | `mq_getattr, mq_setattr` | – | `fstat` |
| Операции IPC | `mq_send, mq_receive` | `sem_wait, sem_post` | `mmap, munmap` |

Основное отличие этих подходов заключается в способах создания идентификаторов:

- в System V используются ключи типа **`key_t`** и функция **`ftok`**;
- в POSIX используются имена, аналогичные именам файлов в файловой системе, но не обязанные соответствовать реальным файлам.

## 6.2. Каналы передачи данных

### 6.2.1. Неименованные каналы

Неименованные каналы – это самая первая форма IPC в UNIX, появившаяся еще в 1973 году. Главным недостатком неименованных каналов является отсутствие имени, вследствие чего они могут использоваться для взаимодействия только родственными процессами. Это было исправлено в UNIX System III (1982) добавлением каналов FIFO, которые еще называются именованными каналами.

Проиллюстрируем использование программных каналов, FIFO и очередей сообщений SysV приложением модели «клиент-сервер» со структурой, приведенной на рис. 6.2. Клиент считывает полное имя

(файла) из стандартного потока ввода и записывает его в канал IPC. Сервер считывает это имя из канала IPC и производит попытку открытия файла на чтение. Если попытка оказывается успешной, сервер считывает файл и записывает его в канал IPC. В противном случае сервер возвращает клиенту сообщение об ошибке. Клиент считывает данные из канала IPC и записывает их в стандартный поток вывода. Если сервер не может считать файл, из канала будет считано сообщение об ошибке. В противном случае будет принято содержимое файла. Две штриховые линии на рис. 6.2 представляют собой канал IPC.


*Рис. 6.2.* Структура приложения модели «клиент-сервер»

Неименованные каналы имеются во всех существующих реализациях и версиях UNIX. Канал создается системным вызовом **pipe** и предоставляет возможность однонаправленной (односторонней) передачи данных:

```
#include <unistd.h>
int pipe(int fd[2]);
```

и возвращает 0 в случае успешного завершения, –1 в случае ошибки. Функция возвращает два файловых дескриптора: **fd[0]** и **fd[1]**, причем первый открыт для чтения, а второй – для записи. На рис. 6.3 изображен канал при использовании его единственным процессом.

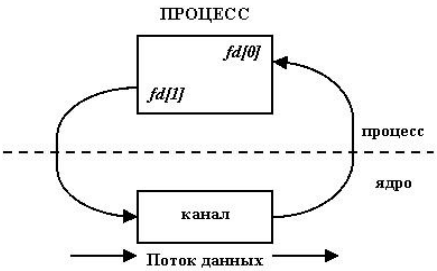



*Рис. 6.3*. Канал при использовании его единственным процессом

Каналы обычно используются для связи между двумя процессами (родительским и дочерним): процесс создает канал, а затем вызывает **`fork`**, создавая свою копию – дочерний процесс (рис. 6.4). Затем родительский процесс закрывает открытый для чтения конец канала, а дочерний, в свою очередь, – открытый на запись конец канала.

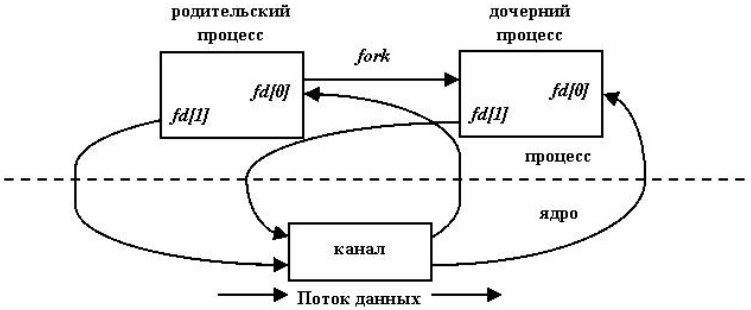



*Рис. 6.4*. Канал при использовании его родственными процессами

Это обеспечивает одностороннюю передачу данных между процессами, как показано на рис. 6.5.

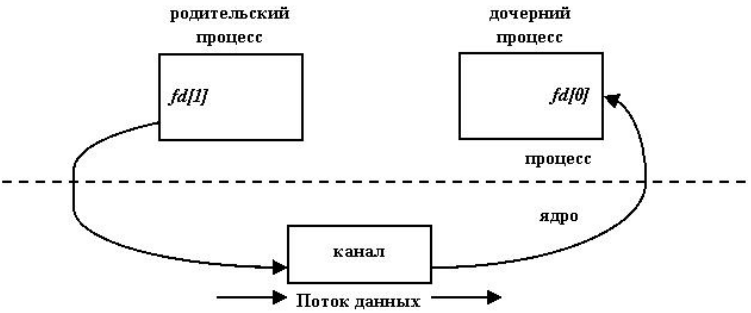



*Рис. 6.5*. Односторонняя передача данных через канал

При вводе команды наподобие **`who | sort | lp`** в интерпретаторе команд UNIX интерпретатор выполняет действия для создания трех процессов с двумя каналами между ними. Интерпретатор также подключает открытый для чтения конец каждого канала к стандартному потоку ввода, а открытый на запись – к стандартному потоку вывода. Созданный при этом канал (конвейер) изображен на рис. 6.6.

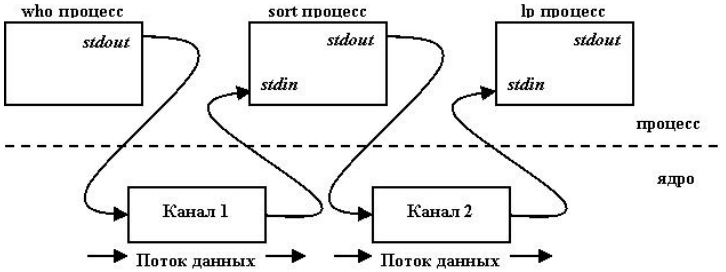



*Рис. 6.6*. Конвейерная передача данных через канал

Все перечисленные каналы были однонаправленными, то есть позволяли передавать данные только в одну сторону. При необходимости передачи данных в обе стороны нужно создавать пару каналов и использовать каждый из них для передачи данных в одну сторону. Этапы создания двунаправленного канала IPC следующие:

1. Создание каналов 1 (**`fd1[0]`** и **`fd1[1]`**) и 2 (**`fd2[0]`** и **`fd2[1]`**).
2. Вызов **`fork`**.
3. Родительский процесс закрывает доступный для чтения конец канала 1 (**`fd1 [0]`**) и доступный для записи конец канала 2 (**`fd2[1]`**).
4. Дочерний процесс закрывает доступный для записи конец канала 1 (**`fd1[1]`**) и доступный для чтения конец канала 2 (**`fd2[0]`**).

Текст программы mainpipe.c доступен на сайте [6] в папке «Pipes». При этом создается структура каналов, изображенная на рис. 6.7.

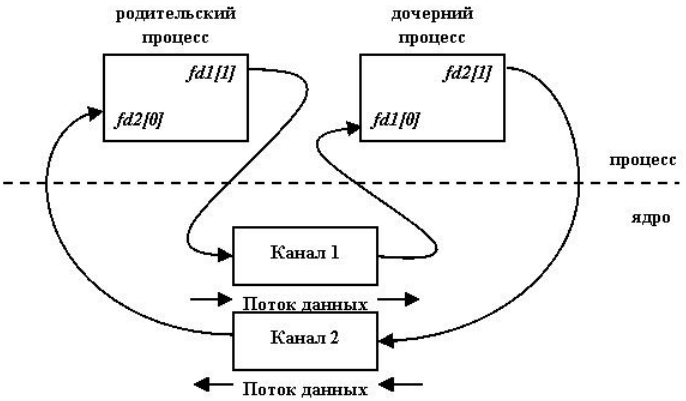



*Рис. 6.7*. Двунаправленная передача данных через каналы

Ниже приведен результат работы программы в случае наличия файла с указанным полным именем и в случае возникновения ошибок:

```
[gun@gun_linux_vm pipes]$ ./mainpipe
/etc/yum.conf   файл из нескольких строк
[main]
cachedir=/var/cache/yum
mirrors=/var/cache/yum/mirrors/
[gun@gun_linux_vm pipes]$ ./mainpipe
/etc/shadow     файл, на чтение которого нет прав
/etc/shadow: can't open  Permission denied
[gun@gun_linux_vm pipes]$ ./mainpipe
/no/such/file   несуществующий файл
/no/such/file: can't open No such file or directory
```

Другим примером использования каналов является имеющаяся в стандартной библиотеке ввода-вывода функция **popen**, которая создает канал и запускает другой процесс, записывающий данные в этот канал или считывающий их из него:

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

Аргумент **command** представляет собой команду интерпретатора. Он обрабатывается программой **sh** (интерпретатор Bourne shell), поэтому для поиска исполняемого файла, вызываемого командой **command**, используется переменная PATH. Канал создается между вызывающим процессом и указанной командой. Возвращаемое функцией **popen** значение представляет собой **NULL** – в случае ошибки, или обычный указатель на тип **FILE**, который может использоваться для ввода или вывода в зависимости от содержимого строки **type**:

- если **type** имеет значение **r**, вызывающий процесс считывает данные, направляемые командой **command** в стандартный поток вывода;
- если **type** имеет значение **w**, вызывающий процесс записывает данные в стандартный поток ввода команды **command**.

Функция **pclose** закрывает стандартный поток ввода-вывода **stream**, созданный командой **popen**, ждет завершения работы программы и возвращает код завершения, принимаемый от интерпретатора или –1 в случае ошибки.

В тексте программы mainpopen.c, доступной на сайте [6] в той же папке, дано еще одно решение задачи с клиентом и сервером, использующее функцию **popen** и программу (утилиту UNIX) **cat**.

Одним из отличий этой реализации от mainpipe.c является отсутствие возможности формировать собственные сообщения об ошибках. Теперь программа зависит от программы **cat**, а выводимые ею сообщения не всегда адекватны. Программа **cat** записывает сообщение об ошибке в стандартный поток сообщений об ошибках (**stderr**), а **popen** с этим потоком не связывается – к создаваемому каналу подключается только стандартный поток вывода.

### 6.2.2. Именованные каналы

Главным недостатком неименованных каналов является невозможность передачи информации между неродственными процессами. Два неродственных процесса не могут создать канал для связи между собой (если не передавать дескриптор).

Аббревиатура FIFO расшифровывается как «first in, first out» – «первым вошел, первым вышел», то есть эти каналы работают как очереди. Именованные каналы в UNIX функционируют подобно неименованным – они позволяют передавать данные только в одну сторону. Однако в отличие от программных каналов каждому каналу FIFO сопоставляется полное имя в файловой системе, что позволяет двум неродственным процессам обратиться к одному и тому же FIFO.

FIFO создается функцией **mkfifo**:

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

Она возвращает 0 при успешном выполнении или –1 при возникновении ошибок. Здесь **pathname** – полное имя файла, которое и будет именем FIFO. Аргумент **mode** указывает битовую маску разрешений доступа к файлу, аналогично второму аргументу команды **open**. В заголовке <sys/stat.h> определены шесть констант, которые могут использоваться для задания разрешений доступа к FIFO.

Функция **mkfifo** действует как **open**, вызванная с аргументом **O_CREAT | O_EXCL**. Это означает, что создается новый канал FIFO или возвращается ошибка **EEXIST**, в случае если канал с заданным

полным именем уже существует. Если не требуется создавать новый канал, вызывайте **open** вместо **mkfifo**. Для открытия существующего канала или создания нового, в том случае, если его еще не существует, вызовите **mkfifo**, проверьте, не возвращена ли ошибка **EEXIST**, и если такое случится, то вызовите функцию **open**.

После создания канал FIFO должен быть открыт на чтение или запись с помощью либо функции **open**, либо **fopen**, но не на чтение и запись, поскольку каналы FIFO могут быть только односторонними.

При записи в программный канал или канал FIFO вызовом **write** данные всегда добавляются к уже имеющимся, а вызов **read** считывает данные, помещенные в программный канал или FIFO первыми. При вызове функции **lseek** для программного канала или FIFO будет возвращена ошибка **ESPIPE**.

Переделаем программу mainpipe.c, чтобы использовать каналы FIFO вместо двух программных каналов. Функции **client** и **server** останутся прежними, отличия появятся только в функции **main**, новый текст которой приведен в файле mainfifo.c, доступном в той же папке сайта.

Константа **FILE_MODE** определена в нем как

```
#define FILE_MODE(S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)
```

Владельцу файла разрешается чтение и запись в него, а группе и прочим пользователям – только чтение. Эти биты разрешений накладываются на маску режима доступа создаваемых файлов процесса.

Изменения по сравнению с примером, в котором использовались программные каналы, следующие:

1. Для создания и открытия программного канала требуется только один вызов – **pipe**. Для создания и открытия FIFO требуется вызов **mkfifo** и последующий вызов **open**.

2. Программный канал автоматически исчезает после того, как будет закрыт последним использующим его процессом. Канал FIFO удаляется из файловой системы только после вызова **unlink**.

Картина аналогична примеру с программными каналами и иллюстрируется рис. 6.8.

Польза от лишнего вызова, необходимого для создания FIFO, следующая: канал FIFO получает имя в файловой системе, что позволяет одному процессу создать такой канал, а другому открыть его, даже если последний не является родственным первому.

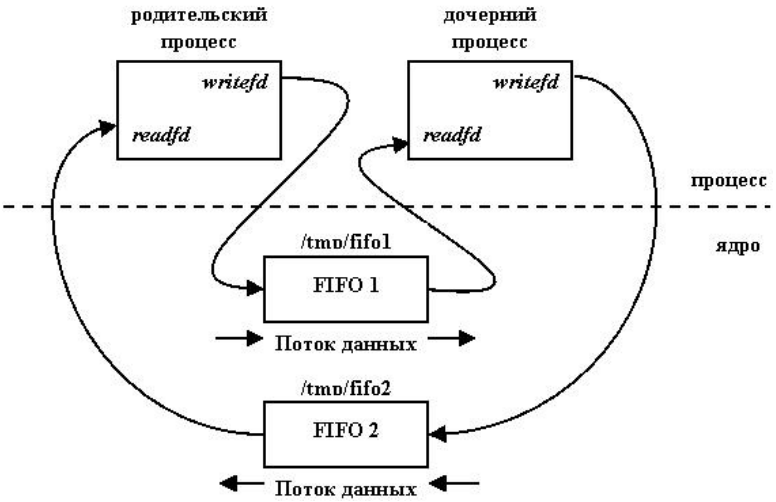



*Рис. 6.8.* Двунаправленная передача данных через каналы FIFO

В программах, некорректно использующих каналы FIFO, могут возникать неочевидные проблемы. Если поменять порядок двух вызовов функции **`open`** в породившем процессе (в файле mainfifo.c), программа перестанет работать. Причина в том, что чтение из FIFO блокирует процесс, если канал еще не открыт на запись другим процессом.

В mainfifo.c клиент и сервер все еще являлись родственными процессами. Переделаем этот пример так, чтобы родство между ними отсутствовало. В файле server_main.c приведен текст программы-сервера. Текст идентичен той части программы из mainfifo.c, которая относилась к серверу. Заголовочный файл fifo.h определяет имена двух FIFO, которые должны быть известны как клиенту, так и серверу. В файле client_main.c приведен текст программы-клиента, которая мало отличается от части программы из mainfifo.c, относящейся к клиенту. Заметим, что именно клиент, а не сервер, удаляет канал FIFO по завершении работы, потому что последние операции с этим каналом выполняются им. Все файлы доступны в той же папке на сайте.

Некоторые свойства именованных и неименованных каналов заслуживают более пристального внимания. Прежде всего, можно сделать дескриптор канала неблокируемым двумя способами.

При вызове **open** указать флаг **O_NONBLOCK**. Например, первый вызов **open** в client_main.c мог бы выглядеть так:

```
writefd = open(FIFO1, O_WRONLY | O_NONBLOCK, 0);
```

Если дескриптор уже открыт, можно использовать **fcntl** для включения флага **O_NONBLOCK**. Эта функция применима для программных каналов, поскольку для них не вызывается функция **open** и нет возможности указать флаг **O_NONBLOCK** при ее вызове. Используя функцию **fcntl**, получим текущий статус файла с помощью **F_GETFL**, затем добавим к нему с помощью побитового логического сложения (**OR**) флаг **O_NONBLOCK** и запишем новый статус командой **F_SETFL**:

```
int flags;
if ( (flags = fcntl(fd, F_GETFL, 0)) < 0)
err_sys("F_GETFL error");
flags != O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
err_sys("F_SETFL error");
```

Будьте аккуратны с программами, которые просто устанавливают требуемый флаг, поскольку при этом сбрасываются все прочие флаги.

Таблица 6.3 иллюстрирует действие флага, отключающего блокировку, при открытии и при чтении данных из пустого программного канала или канала FIFO.

Упомянем несколько дополнительных правил, действующих при чтении и записи данных в программные каналы и FIFO.

1. Попытка считать больше данных, чем в данный момент содержится в канале, возвращает только имеющийся объем данных. Нужно предусмотреть обработку ситуации, в которой функция **read** возвращает меньше данных, чем было запрошено.

2. Если количество байтов, направленных на запись функции **write**, не превышает значения **PIPE_BUF**, то ядро гарантирует атомарность операции записи. Это означает, что если два процесса запишут данные в канал приблизительно одновременно, то в буфер будут помещены сначала все данные от первого процесса, а затем от второго, либо наоборот. Данные от двух процессов при этом не будут смешиваться. Однако если количество байтов превышает значение **PIPE_BUF**, то атомарность операции записи не гарантируется.

Таблица 6.3

**Действие флага `O_NONBLOCK`**

| Операция | Наличие открытых каналов | Блокировка включена (по умолчанию) | Флаг **`O_NONBLOCK`** установлен |
|---|---|---|---|
| Открытие FIFO только для чтения | FIFO открыт на запись | Возвращается код успешного завершения операции | Возвращается код успешного завершения операции |
| Открытие FIFO только для чтения | FIFO не открыт на запись | Блокируется, пока FIFO не будет открыт на запись | Возвращается код успешного завершения операции |
| Открытие FIFO только для записи | FIFO открыт на чтение | Возвращает код успешного завершения операции | Возвращает код успешного завершения операции |
| Открытие FIFO только для записи | FIFO не открыт на чтение | Блокируется до тех пор, пока FIFO не будет открыт на чтение | Возвращает ошибку с кодом **`ENXIO`** |
| Чтение из пустого программного канала или FIFO | Программный канал или FIFO открыт на запись | Блокируется, пока в канал не будут помещены данные или канал не будет закрыт всеми процессами, которыми он был открыт на запись | Возвращает ошибку с кодом **`EAGAIN`** |
| Чтение из пустого программного канала или FIFO | Канал не открыт на запись | **`read`** возвращает 0 (конец файла) | **`read`** возвращает 0 (конец файла) |
| Запись в программный канал или FIFO | Канал открыт на чтение | (См. в тексте) | (См. в тексте) |
| Запись в программный канал или FIFO | Канал не открыт на чтение | Программе посылается сигнал **`SIGPIPE`** | Программе посылается сигнал **`SIGPIPE`** |

Установка флага **`O_NONBLOCK`** не влияет на атомарность (способность выполняться как единое целое) операции записи в канал – она определяется объемом посылаемых данных в сравнении с величиной

**PIPE_BUF**. Однако если для канала отключена блокировка, возвращаемое функцией **write** значение зависит от количества байтов, отправленных на запись, и наличия свободного места в канале. Если количество байтов не превышает величины **PIPE_BUF**, то:

- если в канале достаточно места для записи требуемого количества данных, они будут переданы все сразу;
- если места в программном канале или FIFO недостаточно для записи требуемого объема данных, происходит немедленное завершение работы функции с возвратом ошибки **EAGAIN**.

Если количество байтов превышает значение **PIPE_BUF**, то:

- если в программном канале или FIFO есть место хотя бы для одного байта, ядро передает в буфер столько данных, сколько туда может поместиться, и это количество возвращается функцией **write**;
- если в программном канале или FIFO свободное место отсутствует, происходит завершение работы с возвратом ошибки **EAGAIN**.

При записи в программный канал или FIFO, не открытый для чтения, ядро посылает сигнал **SIGPIPE**:

- если процесс не перехватывает (см. **signal()**) и не игнорирует **SIGPIPE**, выполняется действие по умолчанию – завершение работы процесса;
- если процесс игнорирует сигнал **SIGPIPE** или перехватывает его и возвращается из подпрограммы его обработки, **write** возвращает ошибку с кодом **EPIPE**.

## Вопросы для самопроверки

1. Что такое каналы? В чем отличие неименованных и именованных каналов?
2. В чем разница при использовании для коммуникаций процессов файлов и каналов?
3. Опишите функции создания и использования неименованных каналов.
4. Чем работа с неименованными каналами отличается от работы с файлами?
5. Опишите особенности использования функции **popen()**.
6. Опишите особенности создания и уничтожения именованных каналов в сравнении с неименованными.

7. Чем работа (чтение/запись) с именованными каналами отличается от работы с неименованными?

8. В каких случаях при работе с неименованными и именованными каналами возникают блокировки в программе?

9. Каковы особенности обхода блокировок при работе с неименованными каналами?

10. Каковы особенности поведения функций работы с каналами при обходе блокировок?

## Упражнения

Выполните лабораторную работу № 4 из лабораторного практикума.

# 6.3. System V IPC: очереди сообщений, семафоры, разделяемая память

### 6.3.1. Введение в System V IPC

Заголовочный файл <sys/types.h> определяет тип **key_t** как целое (по меньшей мере 32-разрядное). Значения переменным этого типа обычно присваиваются функцией **ftok**. Функция **ftok** преобразовывает полное имя некоторого существующего файла и целочисленный идентификатор в значение ключа IPC типа **key_t** (IPC key).

```
#include <sys/ipc.h>
key_t ftok(const char *name, int id);
```

Функция использует полное имя файла и младшие 8 бит идентификатора для формирования целочисленного ключа IPC. Функция возвращает ключ IPC либо –1 при возникновении ошибки. Функция предполагает, что для конкретного приложения, использующего IPC, клиент и сервер используют одно и то же полное имя файла. Если клиенту и серверу для связи требуется только один канал IPC, идентификатору можно присвоить, например, значение 1. Если требуется несколько каналов IPC (например, один от сервера к клиенту и один в обратную сторону), идентификаторы должны иметь разные значения: например, 1 и 2. После того как клиент и сервер договорятся о полном имени и идентификаторе, они оба вызывают функцию **ftok** для получения

одинакового ключа IPC. Большинство реализаций функции **ftok** вызывают функцию **stat**, а затем объединяют:

- информацию о файловой системе, к которой относится полное имя **pathname** (поле **st_dev** структуры **stat**);
- номер узла (i-node) в файловой системе (поле **st_ino** структуры **stat**);
- младшие 8 бит идентификатора (не должен равняться нулю!).

Из комбинации этих трех значений получается 32-разрядный ключ. Номер узла (i-node) всегда отличен от нуля, поэтому большинство реализаций определяют константу **IPC_PRIVATE** равной нулю. Если указанное имя файла не существует или недоступно вызывающему процессу, **ftok** возвращает значение –1. Файл, имя которого используется для вычисления ключа, не должен быть одним из тех, которые создаются и удаляются в процессе работы, поскольку каждый раз при создании заново эти файлы получают другой номер узла, а это может изменить ключ, возвращаемый функцией **ftok** при очередном вызове.

Программа ftok.c, доступная для скачивания на сайте [6] в папке «SysV_IPC», принимает полное имя в качестве аргумента командной строки, вызывает функции **stat** и **ftok**, затем выводит значения полей **st_dev** и **st_ino** структуры **stat** и получающийся ключ IPC. Эти три значения выводятся в шестнадцатеричном формате, поэтому легко видеть, как именно ключ IPC формируется из этих двух значений и идентификатора 0x57.

Для каждого объекта IPC, как для обычного файла, в ядре хранится набор информации, объединенной в структуру:

```
struct ipc_perm {
uid_t uid; /* id пользователя владельца */
gid_t gid; /* id группы владельца */
uid_t cuid; /* id создателя */
gid_t cgid; /* id группы создателя */
mode_t mode; /* разрешения чтения-записи */
ulong_t seq; /* Последовательный номер канала */
key_t key; /* ключ IPC */ };
```

Эта структура вместе с другими поименованными константами для функций System V IPC определена в файле <sys/ipc.h>.

Три функции вида **XXXget**, используемые для создания или открытия объектов IPC (табл. 6.1), принимают ключ IPC (типа **key_t**)

в качестве одного из аргументов и возвращают целочисленный идентификатор. У приложения есть две возможности задания ключа (первого аргумента функций **XXXget**).

1. Вызвать **ftok**, передать ей полное имя и идентификатор.

2. Указать в качестве ключа константу **IPC_PRIVATE**, гарантирующую создание нового уникального объекта IPC.

Последовательность действий иллюстрирует рис. 6.9.

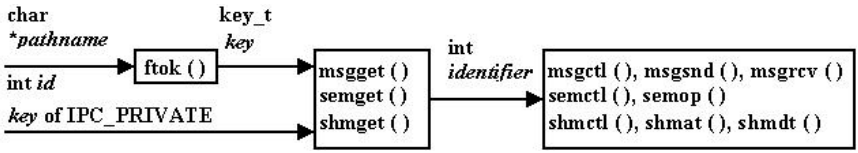



*Рис. 6.9.* Создание и открытие каналов IPC

Все три функции **XXXget** (табл. 6.1) принимают в качестве второго аргумента набор флагов **oflag**, задающий биты разрешений чтения-записи (поле **mode** структуры **ipc_perm**) для объекта IPC и определяющий, создается ли новый объект IPC или производится обращение к уже существующему. Для этого имеются следующие правила.

1. Ключ **IPC_PRIVATE** гарантирует создание уникального объекта IPC. Никакие возможные комбинации полного имени и идентификатора не могут привести к тому, что функция **ftok** вернет в качестве ключа значение **IPC_PRIVATE**.

2. Установка бита **IPC_CREAT** аргумента **oflag** приводит к созданию новой записи для указанного ключа, если она еще не существует. Если запись существует, то возвращается ее идентификатор.

3. Одновременная установка битов **IPC_CREAT** и **IPC_EXCL** аргумента **oflag** приводит к созданию новой записи для указанного ключа только в том случае, если такая запись еще не существует. Если же запись существует, то функция возвращает ошибку **EEXIST**.

4. Комбинация **IPC_CREAT** и **IPC_EXCL** для объектов IPC действует аналогично комбинации **0_CREAT** и **0_EXCL** для функции **open**.

5. Установка бита **IPC_EXCL** без **IPC_CREAT** никакого эффекта не дает.

Логическая диаграмма последовательности действий при открытии объекта IPC изображена на рис. 6.10. В табл. 6.4 этот процесс описан таблично.

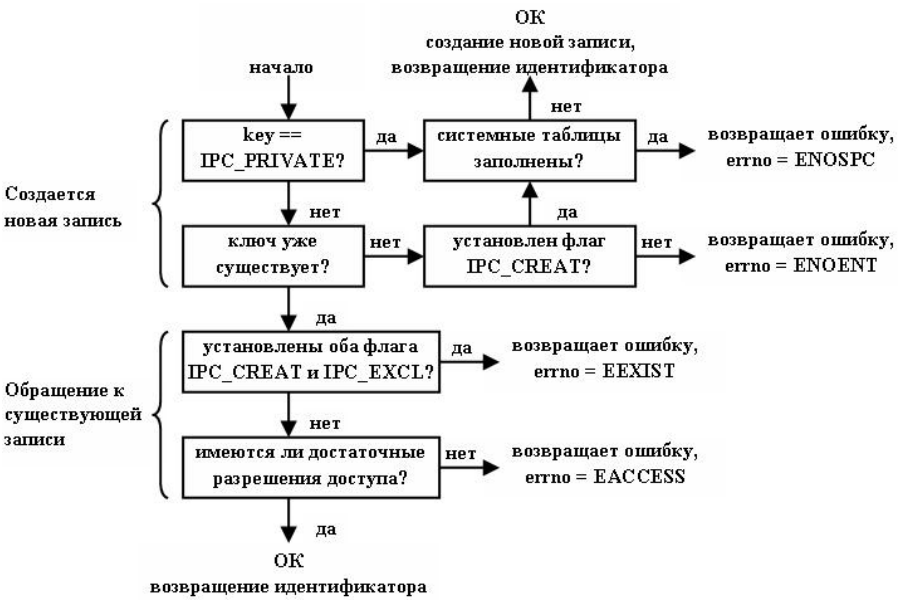



*Рис. 6.10.* Последовательность действий при открытии объекта IPC

Таблица 6.4

**Последовательность действий при открытии объекта IPC**

| Аргумент **oflag** | Ключ не существует | Ключ существует |
|---|---|---|
| Специальные флаги не установлены | Ошибка, **errno** = **ENOENT** | ОК, открытие существующего объекта |
| **IPC_CREAT** | ОК, создается новая запись | ОК, открытие существующего объекта |
| **IPC_CREAT\|IPC_EXCL** | ОК, создается новая запись | Ошибка, **errno** = **EEXIST** |

Обратите внимание, что в средней строке табл. 6.4 для флага **IPC_CREAT** без **IPC_EXCL** мы не получаем никакой информации о том, был ли создан новый объект или получен доступ к существующему объекту. Для большинства приложений характерно создание сервером объекта IPC с указанием **IPC_CREAT** или **IPC_CREAT |**

**IPC_EXCL**. При создании нового объекта IPC с помощью одной из функций **XXXget**, вызванной с флагом **IPC_CREAT**, в структуру **ipc_perm** заносится следующая информация.

1. Часть битов аргумента **oflag** задают значение поля **mode** структуры **ipc_perm**. В табл. 6.5 указаны биты разрешений для трех типов объектов IPC.

2. Поля **cuid** и **cgid** получают значения, равные действующим идентификаторам пользователя и группы вызывающего процесса. Эти два поля называются идентификаторами создателя.

3. Поля **uid** и **gid** устанавливаются равными действующим идентификаторам пользователя и группы вызывающего процесса. Эти два поля называются идентификаторами владельца.

Идентификатор создателя изменяться не может, тогда как идентификатор владельца может быть изменен процессом с помощью вызова функции **XXXctl** для данного типа объекта IPC с командой **IPC_SET**. Эти функции (**XXXctl**) позволяют процессу также изменять биты разрешений доступа (поле **mode**) объекта IPC.

Таблица 6.5

**Биты разрешений для трех типов IPC**

| Число (octal) | Очередь сообщений | Семафор | Разделяемая память | Описание |
|---|---|---|---|---|
| 0400 | **MSG_R** | **SEM_R** | **SHM_R** | Пользователь – чтение |
| 0200 | **MSG_W** | **SEM_A** | **SHM_W** | Пользователь – запись |
| 0040 | **MSG_R>>3** | **SEM_R>>3** | **SHM_R>>3** | Группа – чтение |
| 0020 | **MSG_W>>3** | **SEM_A>>3** | **SHM_W>>3** | Группа – запись |
| 0004 | **MSG_R>>6** | **SEM_R>>6** | **SHM_R>>6** | Прочие – чтение |
| 0002 | **MSG_W>>6** | **SEM_A>>6** | **SHM_W>>6** | Прочие – запись |

Битам соответствуют поименованные константы, показанные в табл. 6.5. Константы определяются в соответствующих (см. табл. 6.1) заголовочных файлах. Суффикс A в **SEM_A** означает «alter» (изменение). Заметим, что в некоторых реализациях UNIX-совместимых ОС перечисленные поименованные константы могут быть не описаны, а потому в целях обеспечения переносимости программ рекомендуется

использовать явное указание битов разрешений в восьмеричной системе счисления.

Когда процесс предпринимает попытку доступа к объекту IPC, производится двухэтапная проверка – при открытии объекта (функция **XXXget**) и каждый раз при обращении к объекту IPC.

1. При установке доступа к существующему объекту IPC с помощью одной из функций **XXXget** производится первичная проверка аргумента **oflag**. Аргумент не должен указывать биты доступа, не установленные в поле **mode** структуры **ipc_perm** (нижний прямоугольник на рис. 6.10). Любой процесс, попытавшийся указать эти биты в аргументе **oflag**, получит ошибку. Любой процесс может пропустить эту проверку, указав аргумент **oflag**, равный 0, если заранее известно о существовании объекта IPC.

2. При любой операции с объектами IPC производится проверка разрешений для процесса, запрашивающего эту операцию. Привилегированному пользователю доступ предоставляется всегда.

3. Если действующий идентификатор пользователя совпадает со значением **uid** или **cuid** объекта IPC и установлен соответствующий бит разрешения доступа в поле **mode** объекта IPC, доступ будет разрешен. Под соответствующим битом разрешения доступа подразумевается бит, разрешающий чтение, если вызывающий процесс запрашивает операцию чтения для данного объекта IPC (например, получение сообщения из очереди), или бит, разрешающий запись, если процесс хочет осуществить ее.

4. Если действующий идентификатор группы совпадает со значением **gid** или **cgid** объекта IPC и установлен соответствующий бит разрешения доступа в поле **mode** объекта IPC, доступ будет разрешен.

5. Если доступ не был разрешен на предыдущих этапах, проверяется наличие соответствующих установленных битов доступа для прочих пользователей.

Структура **ipc_perm** содержит переменную **seq**, в которой хранится порядковый номер канала. Эта переменная представляет собой счетчик, заводимый ядром для каждого объекта IPC в системе. При удалении объекта IPC номер канала увеличивается, а при переполнении – сбрасывается в ноль. Идентификаторы System V IPC устанавливаются для всей системы, а не для процесса.

Если два неродственных процесса используют, например, одну очередь сообщений, то ее идентификатор, возвращаемый функцией

**msgget**, должен иметь одно и то же целочисленное значение в обоих процессах, чтобы они получили доступ к одной и той же очереди.

Такая особенность дает возможность процессу, созданному злоумышленником, попытаться прочесть сообщение из очереди, созданной другим приложением, последовательно перебирая различные идентификаторы и надеясь на существование открытой в текущий момент очереди, доступной для чтения всем. Для исключения такой возможности разработчики средств IPC решили расширить диапазон значений идентификатора путем увеличения значения идентификатора, возвращаемого вызывающему процессу, на количество записей в системной таблице IPC каждый раз, когда происходит повторное использование одной из них.

Счетчик номеров каналов позволяет исключить повторное использование идентификаторов System V IPC через небольшой срок. Это гарантирует, что досрочно завершивший работу и перезапущенный сервер не станет использовать тот же идентификатор.

Программа slot.c, доступная на сайте, выводит первые десять значений идентификаторов, возвращаемых функцией **msgget**. При очередном прохождении цикла **msgget** создает очередь сообщений, а **msgctl** с командой **IPC_RMID** в качестве аргумента удаляет ее. При повторном запуске программы видим наглядную иллюстрацию того, что последовательный номер канала – это переменная, хранящаяся в ядре и продолжающая существовать и после завершения процесса.

В системах, поддерживающих IPC, предоставляются две специальные утилиты: **ipcs**, выводящая различную информацию о свойствах System V IPC, и **ipcrm**, удаляющая очередь сообщений System V, семафор или сегмент разделяемой памяти. Первая из этих функций поддерживает около десятка параметров командной строки, управляющих отображением информации о различных типах IPC. Второй (**ipcrm**) можно задать до шести параметров. Подробную информацию о них можно получить в справочной системе.

Большинству реализаций System V IPC свойственно наличие внутренних ограничений, налагаемых ядром. Это, например, максимальное количество очередей сообщений или ограничение на максимальное количество семафоров в наборе. Их текущие значения в ОС Linux можно вывести на экран, используя команду **sysctl -a | grep kernel.msg**.

### 6.3.2. Очереди сообщений System V IPC

Каждой очереди сообщений System V сопоставляется свой идентификатор. Любой процесс с соответствующими привилегиями может поместить сообщение в очередь, и любой процесс с соответствующими привилегиями может сообщение из очереди считать. Для помещения сообщения в очередь System V не требуется наличия подключенного к ней на считывание процесса.

Ядро хранит информацию о каждой очереди сообщений в виде структуры, определенной в заголовочном файле <sys/msg.h>:

```
struct msqid_ds {
struct ipc_perm msg_perm;//Разрешения чтения
и записи
struct msg *msg_first;    /*указатель на первое
сообщение в очереди */
struct msg *msg_last; /* указатель на последнее
сообщение в очереди */
msglen_t msg_cbytes;  // размер очереди в байтах
msgqnum_t msg_qnum; // количество сообщений
в очереди
msglen_t msg_qbytes; /* максимальный размер очереди
в байтах */
pid_t msg_lspid; /* pid последнего процесса,
вызвавшего msgsnd(); */
pid_t msg_lrpid;// pid последнего msgrcv()
time_t msg_stime; //время отправки последнего
сообщения
time_t msg_rtime; // время последнего считывания
сообщения
time_t msg_ctime; /* время последнего вызова
msgctl(). изменившего одно из полей структуры */
}
```

Представим конкретную очередь сообщений, хранимую ядром как связный список, на рис. 6.11. В этой очереди три сообщения длиной 1, 2 и 3 байта с типами 100, 200 и 300 соответственно.

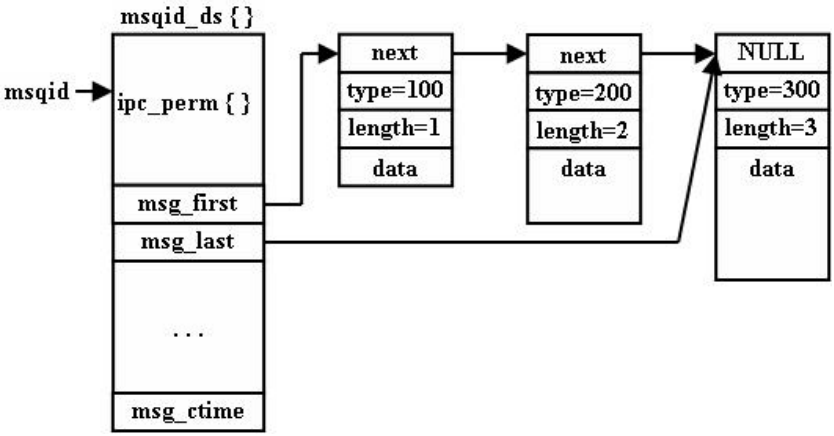



*Рис. 6.11.* Структура очереди сообщений

Для работы с очередью используется ряд функций. Функция **msgget** создает новую очередь сообщений или получает доступ к существующей:

```
#include <sys/msg.h>
int msgget(key_t key, int oflag);
```

Возвращаемое значение – целочисленный идентификатор, используемый тремя другими функциями **msgXXX** для обращения к данной очереди, или –1 в случае ошибки. Идентификатор вычисляется на основе указанного ключа, который формируется функцией **ftok** или может представлять собой константу **IPC_PRIVATE**.

Флаг **oflag** представляет собой комбинацию разрешений чтения-записи. В него можно добавить флаги **IPC_CREAT** или **IPC_CREAT|IPC_EXCL** с помощью логического сложения. При создании новой очереди инициализируются следующие поля структуры **msqid_ds**:

- полям **uid** и **cuid** структуры **msg_perm** присваивается значение действующего идентификатора пользователя вызвавшего процесса, а полям **gid** и **cgid** – действующего идентификатора группы;
- разрешения чтения-записи, указанные в **oflag**, помещаются в **msg_perm.mode**;
- значения **msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime** и **msg_rtime** устанавливаются в 0;

- в **msg_ctime** записывается текущее время;
- в **msg_qbytes** помещается ограничение ядра на размер очереди.

После открытия очереди сообщений с помощью функции **msgget** можно помещать сообщения в эту очередь с помощью функции **msgsnd**:

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t
length, int flag);
```

Функция возвращает 0 в случае успешного завершения и –1 в случае ошибки. Здесь **msqid** представляет собой идентификатор очереди, возвращаемый **msgget**. Указатель **ptr** указывает на структуру следующего шаблона, определенного в <sys/msg.h>:

```
struct msgbuf {
long mtype; /* тип сообщения, должен быть > 0 */
char mtext[1]; /* данные */
};
```

Тип сообщения (**mtype**) должен быть больше нуля, поскольку неположительные типы используются в качестве специальной команды функции **msgrcv**. Большинство приложений затем определяют собственную структуру сообщений, обязательно начинающуюся с поля типа **long**. Например, если приложению нужно передавать сообщения, состоящие из 16-разрядного целого, за которым следует 8-байтовый массив символов, оно может определить свою собственную структуру так:

```
#define MY_DATA 8
typedef struct my_msgbuf {
long mtype; /*тип сообщения */
int16_t mshort; /* начало данных */
char mchar[MY_DATA];
} message;
```

Аргумент **flag** может быть либо 0, либо **IPC_NOWAIT**. В последнем случае он отключает блокировку для **msgsnd**: если для нового сообщения недостаточно места в очереди, возврат из функции происходит немедленно. Это может произойти, если:

- в данной очереди уже имеется слишком много данных (значение **msg_qbytes** в структуре **msqid_ds**);

- во всей системе имеется слишком много сообщений.

Если верно одно из этих условий и установлен флаг **IPC_NOWAIT**, функция **msgsnd** возвращает ошибку с кодом **EAGAIN**. Если флаг **IPC_NOWAIT** не указан, а одно из этих условий выполняется, поток приостанавливается до тех пор, пока:

- для сообщения не освободится достаточно места;
- очередь с идентификатором **msqid** не будет удалена из системы (в этом случае возвращается ошибка с кодом **EIDRM**);
- вызвавший функцию поток не будет прерван перехватываемым сигналом (в этом случае возвращается ошибка с кодом **EINTR**).

Сообщение может быть считано из очереди с помощью функции **msgrcv**:

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *ptr, size_t length,
long type, int flag);
```

Функция возвращает количество данных в сообщении или –1 в случае ошибки. Аргумент **ptr** указывает, куда следует помещать принимаемые данные. Он указывает на поле данных типа **long**, которое предшествует полезным данным. Аргумент **length** задает размер относящейся к полезным данным части буфера, на который указывает **ptr**. Это максимальное количество данных, которое может быть возвращено функцией. Поле типа **long** не входит в эту длину. Аргумент **type** определяет тип сообщения, которое нужно считать из очереди:

- если тип равно 0, возвращается первое сообщение в очереди;
- если тип больше 0, возвращается первое сообщение, тип которого равен указанному;
- если тип меньше 0, возвращается первое сообщение с наименьшим типом, значение которого меньше либо равно модулю **type**.

Рассмотрим пример очереди сообщений, изображенный на рис. 6.11. В этой очереди имеются три сообщения:

- первое сообщение имеет тип 100 и длину 1;
- второе сообщение имеет тип 200 и длину 2;
- третье сообщение имеет тип 300 и длину 3.

Табл. 6.6 показывает, какое сообщение будет возвращено при различных значениях аргумента **type**.

Таблица 6.6

**Определение типа сообщения по аргументу `type`**

| type | Тип возвращаемого сообщения |
|---|---|
| 0 | 100 |
| 100 | 100 |
| 200 | 200 |
| 300 | 300 |
| –100 | 100 |
| –200 | 100 |
| –300 | 100 |

Аргумент **`flag`** указывает, что делать, если в очереди нет сообщения с запрошенным типом. Если установлен бит **`IPC_NOWAIT`**, происходит немедленный возврат из функции **`msgrcv`** с кодом ошибки **`ENOMSG`**. В противном случае вызвавший процесс блокируется до тех пор, пока не произойдет одно из следующего:

- появится сообщение с запрошенным типом;
- очередь с идентификатором **`msqid`** будет удалена из системы (в этом случае будет возвращена ошибка с кодом **`EIDRM`**);
- вызвавший поток будет прерван перехватываемым сигналом (в этом случае возвращается ошибка **`EINTR`**).

В аргументе **`flag`** можно указать бит **`MSG_NOERROR`**. При установке этого бита данные, превышающие объем буфера (аргумент **`length`**), будут просто обрезаться до его размера без возвращения кода ошибки. Если этот флаг не указать, при превышении объемом сообщения аргумента **`length`** будет возвращена ошибка **`E2BIG`**.

В случае успешного завершения работы **`msgrcv`** возвращает количество байтов в принятом сообщении. Оно не включает байты, нужные для хранения типа сообщения (**`long`**), который также возвращается через указатель **`ptr`**.

Функция **`msgctl`** позволяет управлять очередями сообщений:

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buff);
```

Функция возвращает 0 в случае успешного завершения, –1 в случае ошибки. Команд (аргумент **cmd**) может быть три.

- **IPC_RMID** – удаление очереди с идентификатором **msqid** из системы. Все сообщения, имеющиеся в этой очереди, будут утеряны. Для этой команды третий аргумент функции игнорируется.
- **IPC_SET** – установка значений четырех полей структуры **msqid_ds** данной очереди равными значениям соответствующих полей структуры, на которую указывает аргумент **buff**: **msg_perm.uid**, **msg_perm.gid**, **msg_perm.mode**, **msg_qbytes**.
- **IPC_STAT** – возвращает вызвавшему процессу (через **buff**) текущее содержимое структуры **msqid_ds** для очереди **msqid**.

Программа ctl.c, доступная на сайте в папке «SysVmsg», создает очередь сообщений, помещает в нее сообщение с одним байтом информации, вызывает функцию **msgctl** с командой **IPC_STAT**, выполняет команду **ipcs**, используя функцию **system**, а затем удаляет очередь, вызвав функцию **msgctl** с командой **IPC_RMID**.

Поскольку очереди сообщений System V обладают живучестью ядра, мы можем написать несколько отдельных программ для работы с этими очередями и изучить их действие.

Программа msgcreate.c, доступная на сайте [6], создает очередь сообщений. Параметр командной строки **-e** позволяет указать флаг **IPC_EXCL**. Полное имя файла, являющееся обязательным аргументом командной строки, передается функции **ftok**. Получаемый ключ преобразуется в идентификатор функцией **msgget**.

Программа msgsnd.c помещает в очередь одно сообщение заданной длины и типа. В ней создается указатель на структуру **msgbuf** общего вида, а затем выделяется место под буфер записи соответствующего размера вызовом **calloc**. Эта функция инициализирует буфер нулем.

Программа msgrcv.c считывает сообщение из очереди. В командной строке может быть указан параметр **-n**, отключающий блокировку, а параметр **-t** может быть использован для указания типа сообщения в функции **msgrcv**.

Для удаления очереди сообщений мы вызываем функцию **msgctl** с командой **IPC_RMID**, как показано в программе msgrimd.c.

Воспользуемся этими четырьмя программами. Создадим очередь и поместим в нее три сообщения. Файл, указываемый в качестве аргумента **ftok**, обязательно должен существовать. Используем имя су-

ществующего файла при создании очереди сообщений. После этого в очередь поместим три сообщения длиной 1, 2 и 3 байта со значениями типа 100, 200 и 300 (вспомните рис. 6.11). Программа **ipcs** показывает, что в очереди находятся 3 сообщения общим объемом 6 байт.

Продемонстрируем использование аргумента **type** при вызове **msgrcv** для считывания сообщений в произвольном порядке. Вначале запросим сообщение с типом 200, потом сообщение с наименьшим по модулю значением типа, не превышающим 300 (–300), а в потом – первое сообщение в очереди (без указания типа). Запуск **msgrcv** при пустой очереди иллюстрирует действие флага **IPC_NOWAIT**. Если указать положительное значение типа, а сообщений с таким типом в очереди нет и флаг **IPC_NOWAIT** не установлен, то программа зависнет.

После этого очередь можно удалить с помощью программы msgrmid.c или с помощью системной команды **ipcrm -q 100**.

Покажем, что для получения доступа к очереди сообщений System V не обязательно вызывать **msgget**: все, что нужно, – это знать идентификатор очереди сообщений, который легко получить с помощью **ipcs**, и считать разрешения доступа для очереди. msgrcvid.c есть упрощенный вариант программы msgrcv.c. Здесь используется идентификатор очереди, являющийся аргументом командной строки.

Создадим очередь, запишем в нее сообщение, определим идентификатор с помощью **ipcs** и предоставим его программе msgrcvid.c в качестве аргумента командной строки. Удаление очереди и ранее выполнялось по идентификатору.

Наличие поля **type** у каждого сообщения в очереди предоставляет интересные возможности.

1. Поле **type** может использоваться для идентификации сообщений, позволяя нескольким процессам мультиплексировать сообщения в одной очереди. Например, все сообщения от клиентов серверу имеют одно и то же значение типа, тогда как сообщения сервера клиентам имеют различные типы, уникальные для каждого клиента.

2. Поле **type** может использоваться для установки приоритета сообщений. Очереди System V позволяют считывать сообщения в произвольном порядке в зависимости от значений типа сообщений.

3. Можно вызывать **msgrcv** с флагом **IPC_NOWAIT** для считывания сообщений с конкретным типом и немедленного возвращения управления процессу в случае отсутствия таких сообщений.

В табл. 6.7 приведены значения системных ограничений для двух разных реализаций ОС. Первая колонка представляет собой традиционное имя System V для переменной ядра, хранящей это ограничение.

В ОС Linux параметры ядра заданы в файле **/etc/sysctl.conf** в секции **#Controls message queues**. Определить ограничения на очереди можно командой **sysctl -a | grep kernel.msg**.

Таблица 6.7

**Системные ограничения на очереди сообщений**

| Имя | Описание | ASPLinux 9.0 | CentOS 6.9 |
|---|---|---|---|
| **msgmax** | Максимальное количество байтов в сообщении | 8192 | 65 536 |
| **msgmnb** | Максимальное количество байтов в очереди сообщений | 16 384 | 65 536 |
| **msgmni** | Максимальное количество очередей сообщений в системе | 16 | 3746 |

В других ОС, где определить параметры ядра сложнее, можно использовать программу, представленную в файле limits.c, которая определяет ограничения, показанные в табл. 6.7 для текущей системы. Для определения максимально возможного размера сообщения она пытается послать сообщение, в котором будет 65 536 байт данных, и если эта попытка оказывается неудачной, уменьшает этот объем до 65 408 и т. д., пока вызов **msgsnd** не окажется успешным.

Создавая 8-байтовые сообщения, программа смотрит, сколько их поместится в очередь. После определения этого ограничения очередь удаляется и повторяется процедура с 16-байтовыми сообщениями, пока не будет достигнут максимальный размер сообщения.

Системное ограничение на количество одновременно открытых идентификаторов определяется непосредственно созданием очередей до тех пор, пока не произойдет ошибка при вызове **msgget**.

### 6.3.3. Семафоры System V IPC

Идею управления дорожным движением с помощью семафоров можно без особых изменений перенести на управление доступом к данным. Семафор – особая структура, содержащая число больше нуля

или равное нулю и управляющая цепочкой процессов, ожидающих особого состояния на данном семафоре. Семафоры могут использоваться для контролирования доступа к ресурсам: число в семафоре представляет собой количество процессов, которые могут получить доступ к данным.

Каждый раз, когда процесс обращается к данным, значение в семафоре должно быть уменьшено на единицу, но увеличено, когда работа с данными будет прекращена. Если ресурс эксклюзивный, то есть к данным должен иметь доступ только один процесс, то начальное значение в семафоре следует установить единицей.

Семафоры можно использовать и для других целей, например для счетчика ресурсов. В этом случае число в семафоре – количество свободных ресурсов (например, количество свободных ячеек памяти).

Рассмотрим реализацию семафоров в System V. Создает семафор функция **semget:**

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

Здесь **key** – IPC ключ, **nsems** – число семафоров, которое мы хотим создать, и **semflg** – права доступа, закодированные в 12 бит: первые три бита отвечают за режим создания, остальные девять – права на запись и чтение для пользователя, группы и остальных. Более полная информация доступна в **man ipc**.

System V предполагает создание сразу нескольких семафоров, что уменьшает код. Рассмотрим создание семафора на примере программы semop1.c, доступной для скачивания на сайте [6] в разделе «SysVSem».

Управление происходит с помощью функции **semctl**:

```
int semctl(int semid, int semnum, int cmd, ...);
```

Функция выполняет действие **cmd** на наборе семафоров **semid** или на одном семафоре с номером **semnum**. В зависимости от команды функции может понадобиться указать еще один аргумент следующего типа:

```
union semun {
int val;    /* значение для SETVAL */
struct semid_ds *buf;/*буферы для IPC_STAT, IPC_SET */
unsigned short *array;/* массивы для GETALL, SETALL */
struct seminfo *__buf; /* буфер для IPC_INFO */
};
```

Чтобы изменить значение семафора, используют директиву **SETVAL**, новое значение должно быть указано в **semun**:

```
/* создать только один семафор */
  semid = semget(key, 1, 0666 | IPC_CREAT);
/* в семафоре 0 установить значение 1 */
  arg.val = 1;
  semctl(semid, 0, SETVAL, arg);
```

Теперь необходимо удалить семафор, освобождая структуры, использовавшиеся для управления им. Это выполняет директива **IPC_RMID**. Она удаляет семафор и посылает сообщение об этом всем процессам, ожидающим доступа к ресурсу:

```
/* удалить семафор */
  semctl(semid, 0, IPC_RMID);
```

Использовать семафор можно с помощью процедуры semop:

```
int semop(int semid, struct sembuf *sops, unsigned
nsops);
```

Здесь **semid** – идентификатор набора семафоров, **sops** – массив, содержащий операции, которые необходимо произвести, **nsops** – число этих операций. Каждая операция представляется структурой **sembuf**:

```
unsigned short sem_num; short sem_op; short sem_flg;
```

Операции, которые мы можем указать, являются целыми числами и подчиняются трем правилам:

1. **sem_op < 0** – если модуль значения в семафоре больше или равен модулю **sem_op**, то **sem_op** добавляется к значению в семафоре (т. е. значение в семафоре уменьшается). Если модуль **sem_op** больше, то процесс переходит в спящий режим, пока не будет достаточно ресурсов.
2. **sem_op = 0** – процесс спит, пока значение в семафоре не достигнет нуля.
3. **sem_op > 0** – значение **sem_op** добавляется к значению в семафоре, используемый ресурс освобождается.

Рассмотрим применение семафоров на примере задачи писателей – читателя (рис. 6.12).

Имеется буфер, в который несколько процессов $W_1$, ..., $W_n$ могут писать и один процесс R может из него читать. Также операции нельзя выполнять одновременно. Процессы $W_i$ могут писать всегда, когда буфер не полон, а процесс R может читать, когда буфер не пуст. Итак, необходимо три семафора: один управляет доступом к буферу, а два других следят за числом элементов в нем.

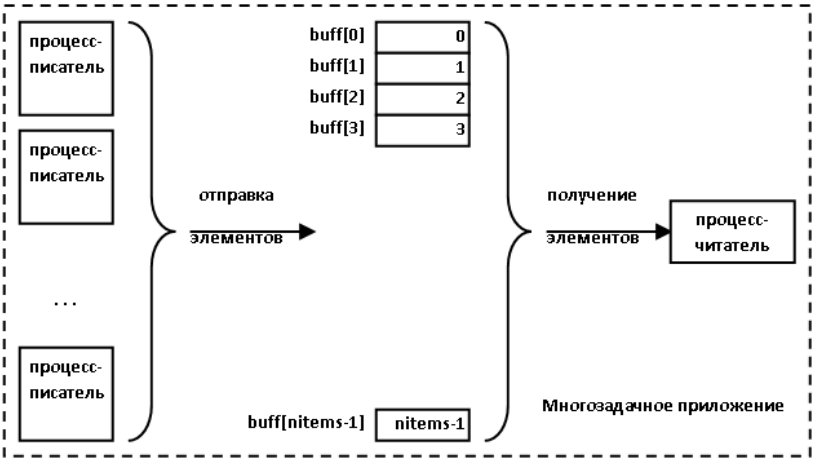



*Рис. 6.12*. Многозадачное приложение «писатели – читатель»

Учитывая, что доступ к буферу должен быть эксклюзивным, первый семафор будет бинарным, в то время как второй и третий будут принимать значения, зависящие от размера буфера. Потребность в двух семафорах связана с особенностью работы функции **semop**. Если, например, процессы $W_i$ уменьшают значение в семафоре, отвечающем за свободное место в буфере, до нуля, то процесс R может увеличивать это значение до бесконечности. Поэтому такой семафор не может указывать на отсутствие элементов в буфере. Текст примера semop2.c доступен на сайте [6]. Прокомментируем наиболее интересные части кода:

```
struct sembuf lock_res = {0, -1, 0};
struct sembuf rel_res = {0, 1, 0};
struct sembuf push[2] = {1, -1, IPC_NOWAIT, 2, 1,
IPC_NOWAIT};
struct sembuf pop[2] = {1, 1, IPC_NOWAIT, 2, -1,
IPC_NOWAIT};
```

Эти четыре строки – действия, которые мы можем производить над семафорами: первые две строки содержат по одному действию каждая, вторые – по два. Первое действие, **lock_res**, блокирует ресурс, уменьшая значение первого (номер 0) семафора на единицу (если значение в семафоре не нуль), а если ресурс уже занят, то процесс ждет. Действие **rel_res** аналогично **lock_res**, только значение в первом семафоре увеличивается на единицу, т. е. убирается блокировка ресурса.

Действия **push** и **pop** – это массивы из двух действий. Первое действие над семафором номер 1, второе – над семафором номер 2; одно увеличивает значение в семафоре, другое уменьшает, но процесс не будет ждать освобождения ресурса: флаг **IPC_NOWAIT** заставляет его продолжить работу, если ресурс заблокирован.

Далее мы инициализируем значения в семафорах: в первом – единицей, так как он контролирует доступ к ресурсу, во втором – длиной буфера (заданной в командной строке), в третьем – нулем (т. е. числом элементов в буфере).

Затем процесс $W_i$ пытается заблокировать ресурс посредством действия **lock_res**; как только это ему удается, он добавляет элемент в буфер посредством действия **push** и выводит сообщение об этом на стандартный вывод.

Если операция не может быть произведена, процесс выводит сообщение о заполнении буфера. В конце процесс освобождает ресурс.

Процесс R ведет себя практически так же, как и процесс $W_i$: блокирует ресурс, производит действие **pop**, освобождает ресурс.

### 6.3.4. Разделяемая память System V IPC

Разделяемая память является самым быстрым средством межпроцессного взаимодействия. После отображения области памяти в адресное пространство процессов, совместно ее использующих, для передачи данных между процессами не требуется участие ядра. Обычно, однако, требуется некоторая форма синхронизации процессов, помещающих данные в разделяемую память и считывающих ее оттуда.

Рассмотрим работу программы чтения файла типа клиент-сервер, примера для иллюстрации различных способов передачи сообщений. В ней сервер считывает данные из входного файла. Данные из файла считываются в ядро, а затем копируются из ядра в память процесса. Сервер составляет сообщение из этих данных и отправляет его, ис-

пользуя именованный или неименованный канал или очередь сообщений. Клиент считывает данные из канала IPC, что обычно требует их копирования из ядра в пространство процесса. Наконец, данные копируются из буфера клиента в выходной файл.

Итого для передачи содержимого файла требуются четыре операции копирования данных. Эти операции копирования между процессами и ядром являются дорогостоящими (более дорогостоящими, чем копирование данных внутри ядра или одного процесса). На рис. 6.13 изображено перемещение данных между клиентом и сервером через ядро.

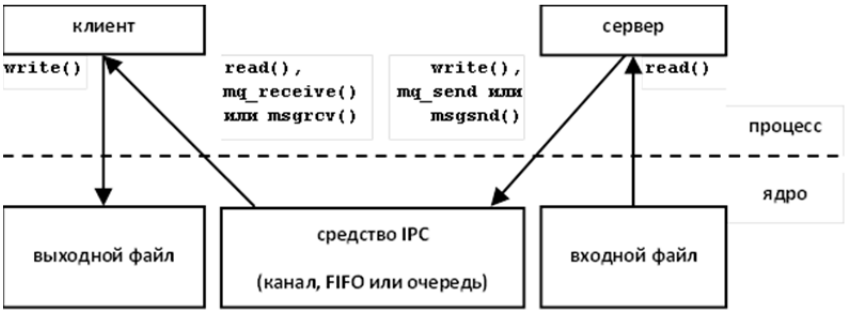



*Рис. 6.13.* Передача содержимого файла через ядро

Недостаток этих форм IPC в том, что для передачи между процессами информация должна пройти через ядро. Разделяемая память дает возможность обойти этот недостаток, поскольку ее использование позволяет двум процессам обмениваться данными через общий участок памяти. Одновременное использование участка памяти во многом аналогично совместному доступу к файлу.

Теперь информация передается между клиентом и сервером в такой последовательности.

1. Сервер получает доступ к объекту разделяемой памяти, используя для синхронизации семафор (например).

2. Сервер считывает данные из файла в разделяемую память. Второй аргумент вызова read указывает на объект разделяемой памяти.

3. После завершения операции считывания клиент уведомляется сервером с помощью семафора.

4. Клиент записывает данные из объекта разделяемой памяти в выходной файл.

Этот сценарий иллюстрирует рис. 6.14.

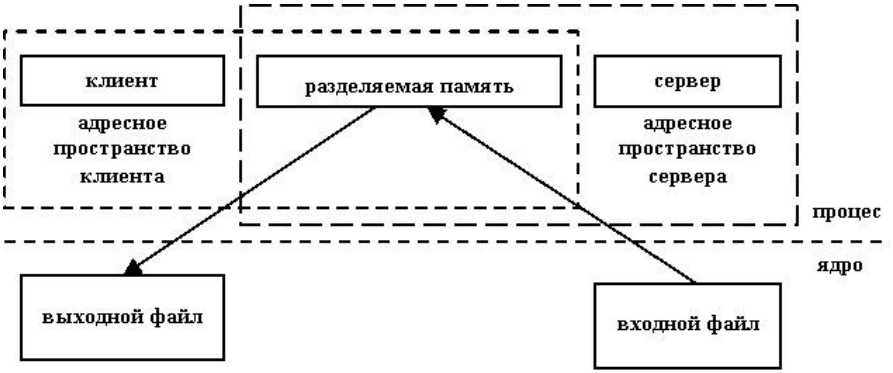



*Рис. 6.14.* Передача содержимого файла через разделяемую память

Из рисунка видно, что копирование данных происходит лишь дважды: из входного файла в разделяемую память и из разделяемой памяти в выходной файл. Два прямоугольника штриховыми линиями подчеркивают, что разделяемая память принадлежит как адресному пространству клиента, так и адресному пространству сервера.

Для каждого сегмента разделяемой памяти ядро хранит структуру **shimd_ds**, определенную в заголовочном файле <sys/shm.h>:

```
struct shimd_ds {
struct ipc_perm shm_perm; /* структура разрешений */
size_t shm_segsz; /* размер сегмента */
pid_t shm_lpid; /* идентификатор последнего процесса*/
pid_t shm_cpid; /*идентификатор процесса-создателя*/
shmatt_t shm_nattch;/*текущее количество
подключений*/
shmat_t shm_cnattch; /* количество подключений
in-core */
time_t shm_atime; /* время последнего подключения */
time_t shm_dtime; /* время последнего отключения */
time_t shm_ctime; /*время последнего изменения */
};
```

Структура **ipc_perm** содержит разрешения доступа к сегменту разделяемой памяти.

С помощью функции **shmget** можно создать новый сегмент разделяемой памяти или получить доступ к существующему сегменту:

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int oflag);
```

Возвращаемое целочисленное значение называется идентификатором разделяемой памяти. Он используется с тремя другими функциями **shmXXX**. В случае ошибки возвращается –1.

Аргумент **key** может содержать значение, возвращаемое функцией **ftok**, или константу **IPC_PRIVATE**. Аргумент **size** указывает размер сегмента в байтах. При создании нового сегмента разделяемой памяти нужно указать ненулевой размер. Если производится обращение к существующему сегменту, аргумент **size** должен быть нулевым. Флаг **oflag** представляет собой комбинацию флагов доступа на чтение и запись. К ним могут быть добавлены с помощью логического сложения флаги **IPC_CREAT** или **IPC_CREAT | IPC_EXCL**. Новый сегмент инициализируется нулями.

Функция **shmget** создает или открывает сегмент разделяемой памяти, но не дает вызвавшему процессу доступа к нему. Для подключения сегмента разделяемой памяти к адресному пространству процесса предназначена функция **shmat**:

```
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int
flag);
```

Аргумент **shmid** – это идентификатор разделяемой памяти, возвращенный **shmget**. Функция **shmat** возвращает адрес начала области разделяемой памяти в адресном пространстве вызвавшего процесса или –1 в случае ошибки. Правила формирования адреса таковы:

- если аргумент **shmaddr** представляет собой нулевой указатель, система сама выбирает начальный адрес, – это рекомендуемый (и обеспечивающий наилучшую совместимость) метод;
- если **shmaddr** отличен от нуля, возвращаемый адрес зависит от того, был ли указан флаг **SHM_RND** (в аргументе **flag**);
- если флаг **SHM_RND** не указан, разделяемая память подключается непосредственно с адреса, указанного аргументом **shmaddr**;
- если флаг **SHM_RND** указан, сегмент разделяемой памяти подключается с адреса, указанного аргументом **shmaddr**, округленного в

меньшую сторону до кратного константе **SHMLBA**. Аббревиатура LBA означает lower boundary address – нижний граничный адрес.

По умолчанию при наличии разрешений сегмент подключается для чтения и записи. В аргументе **flag** можно указать константу **SHM_RDONLY**, которая позволит установить доступ только на чтение.

После завершения работы с сегментом разделяемой памяти его следует отключить вызовом **shmdt**:

```
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

Функция возвращает 0 в случае успешного завершения, –1 в случае ошибки. Эта функция не удаляет сегмент разделяемой памяти. Удаление осуществляется функцией **shmctl** с командой **IPC_RMID**. При завершении работы процесса все сегменты, которые не были отключены им явно, отключаются автоматически, но не удаляются.

Функция **shmctl** позволяет выполнять различные операции с сегментом разделяемой памяти:

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds
*buff);
```

Функция возвращает 0 в случае успешного завершения, –1 в случае ошибки. Команд (значений аргумента **cmd***)* может быть три.

1. **IPC_RMID** – удаление сегмента разделяемой памяти с идентификатором **shmid** из системы.

2. **IPC_SET** – установка значений полей структуры **shmid_ds** для сегмента разделяемой памяти равными значениям соответствующих полей структуры, на которую указывает аргумент **buff***:* **shm_perm.uid**, **shm_perm.gid**, **shm_perm.mode**. Значение поля **shm_ctime** устанавливается равным текущему системному времени. Применяется для смены прав доступа к объекту, передачи прав владения другому пользователю через установку значений полей структуры **shm_perm**, вложенной в структуру **shimd_ds**.

3. **IPC_STAT** – возвращает (через аргумент **buff**) текущее значение структуры **shmid_ds** для указанного сегмента разделяемой памяти. Применяется, в частности, для определения размера сегмента (поле **shm_segsz** структуры **shimd_ds**), если сегмент создан в другом процессе.

Приведем несколько примеров простых программ, иллюстрирующих работу с разделяемой памятью System V.

Программа shmget.c, текст которой доступен на сайте в разделе «SysVshm», создает сегмент разделяемой памяти, принимая из командной строки полное имя и длину сегмента. Вызов **shmget** создает сегмент разделяемой памяти указанного размера. Полное имя, передаваемое в качестве аргумента командной строки, преобразуется в ключ IPC System V вызовом **ftok**. Если указан параметр **-e**, наличие существующего сегмента с тем же именем приведет к возвращению ошибки. Если мы знаем, что сегмент уже существует, то в командной строке должна быть указана нулевая длина. После этого программа завершает работу. Разделяемая память System V обладает живучестью ядра, поэтому сегмент не удаляется.

Тривиальная программа shmrmid.c вызывает функцию **shmctl** с командой **IPC_RMID** для удаления сегмента разделяемой памяти из системы.

Программа shmwrite.c заполняет сегмент разделяемой памяти последовательностью значений 0, 1, 2, …, 254, 255, 0, 1… Сегмент открывается вызовом **shmget** и подключается вызовом **shmat**. Его размер может быть получен вызовом **shmctl** с командой **IPC_STAT**.

Программа shmread.c проверяет последовательность значений, записанную в разделяемую память программой shmwrite.c.

Проиллюстрируем применение этих простых программ. Создадим в системе CentOS сегмент разделяемой памяти длиной 1234 байта. Для идентификации сегмента используем полное имя исполняемого файла **shmget**. Это имя будет передано функции **ftok**. Имя исполняемого файла часто используется в качестве уникального идентификатора:

```
[root@gun_linux_vm]# ./shmget shmget 1234
[root@gun_linux_vm]# ipcs –m
```

Программу **ipcs** запускаем для того, чтобы убедиться, что сегмент разделяемой памяти был создан и не был удален по завершении программы **shmget**. Количество подключений равно нулю.

Запустим программу **shmwrite**, чтобы заполнить содержимое разделяемой памяти последовательностью значений. Проверим содержимое сегмента разделяемой памяти программой **shmread** и удалим сегмент:

```
[root@gun_linux_vm]# ./shmwrite shmget
```

```
[root@gun_linux_vm]# ./shmread shmget
[root@gun_linux_vm]# ./shmrmid shmget
[root@gun_linux_vm]# ipcs –m
```

Программа **ipcs** позволит убедиться, что сегмент разделяемой памяти действительно был удален.

В табл. 6.8 приведены значения ограничений ядра для разных реализаций. В первом столбце приведены традиционные для System V имена переменных ядра, в которых хранятся эти ограничения.

Таблица 6.8

**Системные ограничения на разделяемую память**

| Имя | Описание | ASPLinux 9.0 | CentOS 6.9 |
|---|---|---|---|
| shmmax | Максимальный размер сегмента, байт | 33 554 432 | 68 719 476 736 |
| shmmnb | Минимальный размер сегмента, байт | 1 | 1 |
| shmmni | Максимальное количество идентификаторов в системе | 4096 | 4096 |
| shmseg | Максимальное количество сегментов, подключенных к процессу | 4096 | 4096 |

Программа limits.c из той же папки сайта [6] определяет значения четырех ограничений, приведенных в табл. 6.8. Запустив эту программу в ASPLinux, увидим:

```
[root@gun_linux_vm]# ./limits
4096 identifiers open at once
4096 shared memory segments attached at once
minimum size of shared memory segment=1
max size of shared memory segment=33554432
```

Эти ограничения в ОС Linux можно проверить командой **sysctl –a | grep kernel.shm*.**

# Вопросы для самопроверки

**1. System V IPC**

1.1. Зачем нужна функция **`ftok`**? Что можно применить вместо нее?

1.2. Какие существуют флаги создания объектов IPC? Как они сочетаются?

1.3. Какие ошибки могут возникнуть при создании объектов IPC?

1.4. В каких форматах можно задать биты разрешений доступа объектов IPC?

1.5. Как можно изменить владельца и задать биты разрешений доступа объектов IPC?

1.6. Каков порядок проверки прав доступа к объектам IPC?

1.7. Какие существуют утилиты для работы с объектами IPC?

**2. Очереди сообщений System V IPC**

2.1. Опишите структуру, в которой ядро хранит информацию об очередях сообщений.

2.2. Когда и как происходит инициализация структуры, в которой ядро хранит информацию об очереди сообщений?

2.3. Каковы особенности функции для отправки сообщения в очередь сообщений?

2.4. В каких ситуациях возможны блокировки при записи сообщения в очередь и до каких пор они длятся?

2.5. Каковы особенности функции для чтения сообщения из очереди сообщений?

2.6. В каких ситуациях возможны блокировки при чтении сообщения из очереди и до каких пор они длятся?

2.7. Какие существуют команды управления очередями сообщений?

2.8. Какова область видимости и время жизни очереди сообщений?

2.9. Какие особенности очередей сообщений можно использовать для мультиплексирования сообщений между несколькими процессами?

2.10. Какие особенности очередей сообщений можно использовать для параллельной обработки сообщений от нескольких процессов?

2.11. Каковы системные ограничения на параметры очередей сообщений и как их определить?

**3. Семафоры System V IPC**

3.1. Что представляют собой семафоры и каковы варианты их использования?

3.2. Каковы особенности создания семафоров (по сравнению с другими объектами IPC)?

3.3. Каковы команды управления семафорами и какой дополнительный аргумент функции **`semctl`** может для них потребоваться?

3.4. Каковы особенности функции использования семафоров **`semop`**?

3.4. В каких случаях при использовании семафоров возможны блокировки работы программы? Как эти блокировки обойти?

**4. Разделяемая память System V IPC**

4.1. Каковы особенности разделяемой памяти как средства IPC?

4.2. Опишите структуру, в которой ядро хранит информацию о разделяемой памяти.

4.3. Каковы особенности создания нового сегмента разделяемой памяти и получения доступа к уже существующему сегменту?

4.4. Требуется ли инициализация нового сегмента (как семафора)? Почему?

4.5. Каковы особенности подключения (**`shmat`**) к сегменту разделяемой памяти?

4.6. Когда и почему требуется отключение (**`shmdt`**) от сегмента разделяемой памяти?

4.7. Каковы команды управления (**`shmctl`**) сегментом разделяемой памяти? В каких случаях они применяются?

4.8. Какова область видимости и время жизни сегментов разделяемой памяти?

4.9. Каковы системные ограничения на параметры сегментов разделяемой памяти и как их определить?

## Упражнения

1. Выполните лабораторную работу № 5 из лабораторного практикума.

2. Выполните лабораторную работу № 6 из лабораторного практикума.

## 6.4. POSIX IPC: очереди сообщений, семафоры, разделяемая память

### 6.4.1. Введение в POSIX IPC

Из имеющихся типов IPC следующие три могут быть отнесены к POSIX IPC, т. е. к методам взаимодействия процессов, соответствующим стандарту POSIX:

- очереди сообщений POSIX;
- семафоры POSIX;
- разделяемая память POSIX.

Эти три вида IPC обладают общими свойствами, и для работы с ними используются похожие функции. В этом параграфе речь пойдет об общих требованиях к полным именам файлов, используемых в качестве идентификаторов, о флагах, указываемых при открытии или создании объектов IPC, и о разрешениях на доступ к ним.

Полный список функций, используемых для работы с данными типами IPC, приведен в табл. 6.9.

Таблица 6.9

**Последовательность действий при открытии объекта IPC**

| | Очереди сообщений | Семафоры | Общая память |
|---|---|---|---|
| Заголовочный файл | **<mqueue.h>** | **<semaphore.h>** | **<sys/mman.h>** |
| Функции для создания, открытия и удаления | **mq_open, mq_close, mq_unlink** | **sem_open, sem_close, sem_unlink, sem_init, sem_destroy** | **shm_open, shm_unlink** |
| Операции управления | **mq_getattr, mq_setattr** | | **ftruncate, fstat** |
| Операции IPC | **mq_send, mq_receive, mq_notify** | **sem_wait, sem_trywait, sem_post, sem_getvalue** | **mmap, munmap** |

Три типа IPC стандарта POSIX имеют идентификаторы (имена), соответствующие этому стандарту. Имя IPC передается в качестве первого аргумента одной из трех функций: **mq_open**, **sem_open** и **shm_open**, причем оно не обязательно должно соответствовать реальному файлу в файловой системе. Стандарт POSIX.1 накладывает на имена IPC следующие ограничения.

1. Имя должно соответствовать существующим требованиям к именам файлов (не превышать в длину **PATHMAX** (см. **/usr/include/limits.h**) байтов, включая завершающий символ с кодом 0).

2. Если имя начинается со слеша (/), вызов любой из этих функций приведет к обращению к одной и той же очереди. В противном случае результат зависит от реализации ОС.

3. Интерпретация дополнительных слешей в имени зависит от реализации ОС.

Таким образом, для лучшей переносимости имена должны начинаться со слеша (/) и не содержать в себе дополнительных слешей.

В современных Linux-системах объекты POSIX создаются в виртуальной файловой системе. Эта файловая система может быть смонтирована суперпользователем, например для очередей сообщений это делается следующими командами:

```
[root@CentOS]# mkdir /dev/mqueue
[root@CentOS]# mount -t mqueue none /dev/mqueue
```

После монтирования объекты POSIX можно просматривать и удалять обычными утилитами для работы с файлами **ls**, **cat**, **rm**:

```
[gun@CentOS]$ ls -l /dev/mqueue
-rw-------. 1 gun gun 80 Янв 30 14:25 tmp.123
```

Все три функции, используемые для создания или открытия объектов IPC – **mq_open**, **sem_open** и **shm_open** – принимают специальный флаг **oflag** в качестве второго аргумента. Он определяет параметры открытия запрашиваемого объекта аналогично второму аргументу стандартной функции **open**. Все константы, из которых можно формировать этот аргумент, приведены в табл. 6.10.

Первые три строки описывают тип доступа к создаваемому объекту: только чтение, только запись, чтение и запись. Очередь сообщений может быть открыта в любом из трех режимов доступа, тогда как для семафора указание этих констант не требуется (для любой операции с

семафором требуется доступ на чтение и запись). Наконец, объект разделяемой памяти не может быть открыт только на запись.

Таблица 6.10

**Константы, используемые при создании и открытии объектов IPC**

| Описание | mq_open | sem_open | shm_open |
|---|---|---|---|
| Только чтение | O_RDONLY | | O_RDONLY |
| Только запись | O_WRONLY | | |
| Чтение и запись | O_RDWR | | O_RDWR |
| Создать, если не существует | O_CREAT | O_CREAT | O_CREAT |
| Исключающее создание | O_EXCL | O_EXCL | O_EXCL |
| Без блокировки | O_NONBLOCK | | |
| Сократить (truncate) существующий | | | O_TRUNC |

Указание прочих флагов из табл. 6.10 не является обязательным.

При создании новой очереди сообщений, семафора или сегмента разделяемой памяти требуется указание по крайней мере одного дополнительного аргумента, определяющего режим. Этот аргумент указывает биты разрешения на доступ к файлу и формируется путем побитового логического сложения констант из табл. 6.11.

Эти константы определены в заголовке <sys/stat.h>. Указанные биты разрешений изменяются наложением маски режима создания файлов для данного процесса.

Как и со вновь созданным файлом, при создании очереди сообщений, семафора или сегмента разделяемой памяти им присваивается идентификатор пользователя, соответствующий действующему идентификатору пользователя процесса. Идентификатор группы пользователей семафора или сегмента разделяемой памяти устанавливается равным действующему групповому идентификатору процесса или групповому идентификатору, установленному по умолчанию для системы в целом. Групповой идентификатор очереди сообщений всегда устанавливается равным действующему групповому идентификатору процесса.

Таблица 6.11

**Константы режима доступа при создании нового объекта IPC**

| Константа | Описание |
|---|---|
| **S_IRUSR** | Владелец – чтение |
| **S_IWUSR** | Владелец – запись |
| **S_IRGRP** | Группа – чтение |
| **S_IWGRP** | Группа – запись |
| **S_IROTH** | Прочие – чтение |
| **S_IWOTH** | Прочие – запись |

Флаг **O_CREAT** обеспечивает создание очереди сообщений, семафора или сегмента разделяемой памяти, если таковой еще не существует.

Флаг **O_EXCL**, если он указан одновременно с **O_CREAT**, требует создания новой очередь сообщений, семафора или объекта разделяемой памяти только в том случае, если таковой не существует. Если объект уже существует и указаны флаги **O_CREAT | O_EXCL**, возвращается ошибка **EEXIST**.

Проверка существования очереди сообщений, семафора или сегмента разделяемой памяти и его создание (в случае отсутствия) должны производиться только одним процессом. Два аналогичных флага имеются и в System V IPC, они описаны в параграфе 6.3.1.

Флаг **O_NONBLOCK** создает очередь сообщений без блокировки. Блокировка обычно устанавливается для считывания из пустой очереди или записи в полную очередь. Об этом более подробно рассказано в параграфе, посвященном функциям **mq_send** и **mq_receive.**

Флаг **O_TRUNC** в случае, если уже существующий сегмент общей памяти открыт на чтение и запись, указывает на необходимость сократить его размер до 0.

На рис. 6.15 показана логическая диаграмма последовательности логических операций при открытии объекта IPC. Другой (текстовый) подход к алгоритму на рис. 6.15 представлен в табл. 6.12.

Обратите внимание, что в средней строке табл. 6.12, где задан только флаг **O_CREAT**, мы не получаем никакой информации о том, был ли создан новый объект или открыт существующий.

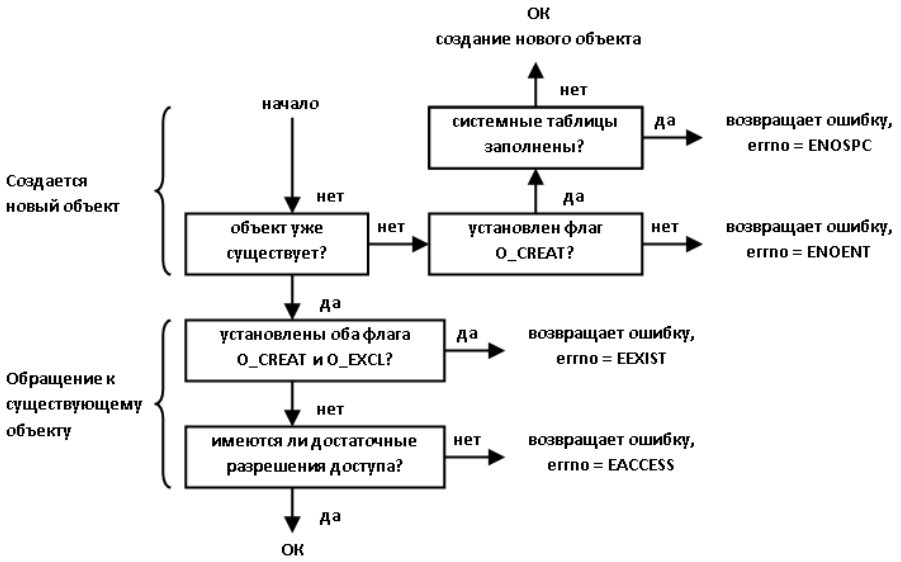



*Рис. 6.15.* Логика открытия объекта POSIX IPC

Таблица 6.12

**Последовательность действий при открытии объекта IPC**

| Аргумент **oflag** | Объект не существует | Объект уже существует |
|---|---|---|
| Нет специальных флагов | Ошибка, **errno=ENOENT** | OK, открывается существующий объект |
| **O_CREAT** | OK, создается новый объект | OK, открывается существующий объект |
| **O_CREAT\|O_EXCL** | OK, создается новый объект | Ошибка, **errno=EEXIST** |

Новая очередь сообщений, именованный семафор или сегмент разделяемой памяти создается функциями **mq_open**, **sem_open** и **shm_open** при условии, что аргумент **oflag** содержит константу **O_CREAT**. Согласно табл. 6.11, любому из данных типов IPC присваиваются определенные права доступа, аналогичные разрешениям доступа к файлам в UNIX.

При открытии существующей очереди сообщений, семафора или сегмента разделяемой памяти теми же функциями (в случае, когда не указан флаг **O_CREAT** или указан **O_CREAT** без **O_EXCL** и объект уже существует) производится проверка разрешений.

1. Проверяются биты разрешений, присвоенные объекту IPC при создании.

2. Проверяется запрошенный тип доступа (**O_RDONLY**, **O_WRONLY**, **O_RDWR**).

3. Проверяется действующий идентификатор пользователя вызывающего процесса, действующий групповой идентификатор процесса.

Большинством систем UNIX производятся следующие конкретные проверки:

1. Если действующий идентификатор пользователя для процесса есть 0 (привилегированный пользователь), доступ будет разрешен.

2. Если действующий идентификатор пользователя процесса совпадает с идентификатором владельца объекта IPC и если соответствующий бит разрешения для пользователя установлен, доступ разрешен, иначе в доступе отказывается. Под соответствующим битом разрешения подразумевается, например, бит разрешения на чтение, если процесс открывает объект только для чтения. Если процесс открывает объект для записи, должен быть установлен соответственно бит разрешения на запись для владельца.

3. Если действующий идентификатор группы процесса совпадает с групповым идентификатором объекта IPC и если соответствующий бит разрешения для группы установлен, доступ будет разрешен, иначе в доступе отказывается.

4. Если соответствующий бит разрешения доступа для прочих пользователей установлен, доступ будет разрешен, иначе в доступе будет отказано.

Эти четыре проверки производятся в указанном порядке. Следовательно, если процесс является владельцем объекта IPC (шаг 2), доступ разрешается или запрещается на основе одних только разрешений пользователя (владельца). Разрешения группы при этом не проверяются. Аналогично, если процесс не является владельцем объекта IPC, но принадлежит к нужной группе, то доступ разрешается или запрещается на основе разрешений группы – разрешения для прочих пользователей при этом не проверяются.

### 6.4.2. Очереди сообщений POSIX IPC

Очередь сообщений можно рассматривать как связный список сообщений. Процессы с соответствующими разрешениями могут помещать сообщения в очередь, а процессы с другими соответствующими разрешениями могут извлекать их оттуда. Каждое сообщение представляет собой запись, и каждому сообщению его отправителем присваивается приоритет. Для записи сообщения в очередь не требуется наличия ожидающего его процесса. Это отличает очереди сообщений от программных каналов и FIFO, в которые нельзя произвести запись, пока не появится считывающий данные процесс.

Процесс может записать в очередь какие-то сообщения, после чего они могут быть получены другим процессом в любое время, даже если первый завершит свою работу. Говорят, что очереди сообщений обладают живучестью ядра. Это также отличает их от программных каналов и FIFO.

Главные отличия очереди сообщений стандарта POSIX от стандарта System V заключаются в следующем:

- операция считывания из очереди сообщений POSIX всегда возвращает самое старое сообщение с наивысшим приоритетом, тогда как из очереди System V можно считать сообщение с произвольно указанным типом;
- очереди сообщений Posix позволяют отправить сигнал или запустить программный процесс при помещении сообщения в пустую очередь, тогда как для очередей System V ничего подобного не предусматривается.

Каждое сообщение в очереди состоит из следующих частей:

- приоритет (беззнаковое целое, POSIX) либо тип сообщения (длинное целое, System V);
- длина полезной части сообщения, которая может быть нулевой;
- собственно данные (если длина сообщения отлична от 0).

Этим очереди сообщений отличаются от программных каналов и FIFO. Последние две части сообщения представляют собой байтовые потоки, в которых отсутствуют границы между сообщениями и никак не указывается их тип. На рис. 6.16 показан возможный вид очереди сообщений.

Реализация очередей сообщений POSIX через связный список предполагает, что его заголовок содержит два атрибута очереди: максимально допустимое количество сообщений в ней и максимальный размер сообщения.

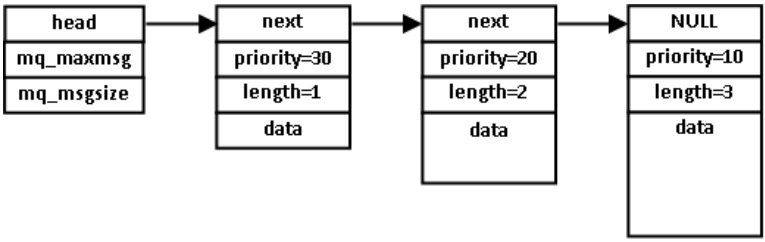



*Рис. 6.16.* Очередь сообщений POSIX, содержащая три сообщения

Поскольку все объекты IPC обладают живучестью ядра, можно написать программу, создающую очередь сообщений POSIX, а потом написать другую программу, которая помещает сообщение в такую очередь, а потом еще одну, которая будет считывать сообщения из очереди. Помещая в очередь сообщения с различным приоритетом, можно увидеть, в каком порядке они будут возвращаться функцией **mq_receive**. Но вначале рассмотрим функции открытия, закрытия и удаления очередей сообщений POSIX: **mq_open, mq_close, mq_unlink.**

### Функции mq_open, mq_close, mq_unlink

Функция **mq_open** создает новую очередь сообщений либо открывает существующую:

```
#include <mqueue.h>
mqd_t mq_open(char *name, int oflag, /* mode_t
mode, struct mq_attr *attr */ );
```

Функция возвращает дескриптор очереди в случае успешного завершения или –1 в противном случае.

Требования к аргументу **name** описаны выше.

Аргумент **oflag** может принимать одно из следующих значений: **O_RDONLY**, **O_WRONLY**, **O_RDWR** в сочетании (логическое сложение) с **O_CREAT**, **O_EXCL**, **O_NONBLOCK**. Все эти флаги описаны в предыдущем параграфе.

При создании новой очереди (указан флаг **O_CREAT** и очередь сообщений еще не существует) требуется указание аргументов **mode** и **attr**. Возможные значения аргумента **mode** приведены в табл. 6.11. Аргумент **attr** позволяет задать некоторые атрибуты очереди. Если в

качестве этого аргумента задать нулевой указатель, очередь будет создана с атрибутами по умолчанию. Сами атрибуты описаны ниже.

Возвращаемое функцией **mq_open** значение называется дескриптором очереди сообщений, но оно не обязательно должно быть (и скорее всего, не является) небольшим целым числом, как дескриптор файла или программного сокета. В большинстве реализаций дескрипторы трактуются как указатели на структуру. Название «дескриптор» было дано им по ошибке. Это значение используется в качестве первого аргумента оставшихся семи функций для работы с очередями сообщений.

Открытая очередь сообщений закрывается функцией **mq_close**:

```
#include <mqueue.h>
int mq_close(mqd_t mqdes );
```

Функция возвращает 0 в случае успешного завершения или –1 в случае ошибки.

По действию эта функция аналогична **close** для открытого файла: вызвавший функцию процесс больше не может использовать дескриптор, но очередь сообщений из системы не удаляется. При завершении процесса все открытые очереди сообщений закрываются, как если бы для каждой был сделан вызов **mq_close**.

Для удаления из системы имени (**name**), которое использовалось в качестве аргумента при вызове **mq_open**, нужно использовать функцию **mq_unlink**:

```
#include <mqueue.h>
int mq_unlink(char *name );
```

Функция возвращает 0 в случае успешного завершения и –1 в случае ошибки.

Для очереди сообщений (как и для файла) ведется подсчет числа процессов, в которых она открыта в данный момент, и по действию эта функция аналогична **unlink** для файла: имя (**name**) может быть удалено из системы, даже пока число подключений к очереди отлично от нуля, но удаление очереди (в отличие от удаления имени из системы) не будет осуществлено до того, как очередь будет закрыта последним использовавшим ее процессом. Очереди сообщений POSIX продолжают существовать, храня все имеющиеся в них сообщения, даже если нет процессов, в которых они были бы открыты. Очередь существует, пока она не будет удалена явно с помощью **mq_unlink**.

Если очередь сообщений реализована через отображаемые в память файлы (как будет показано далее), она может обладать живучестью файловой системы, но это не является обязательным и рассчитывать на это нельзя.

Пример создания очереди сообщений POSIX приведен в программе mqcreate1.c, доступной в разделе «PosixMsg» сайта [6]. Сборка программы потребует ключа **gcc -lrt**, поскольку библиотека librt.so содержит функции, обеспечивающие большинство интерфейсов, описанных в POSIX.1b Realtime Extension:

```
[gun@CentOS]$ gcc -Wall -o mqcreate1 mqcreate1.c -lrt
```

В командной строке при запуске этой программы можно указать параметр **-e**, требующий создания эксклюзивной очереди. Программа вызывает функцию **mq_open**, указывая ей в качестве имени IPC полученный из командной строки параметр. Ниже приведен результат работы программы в CentOS 6.9.

*Очередь успешно создается*:

```
[gun@CentOS]$ ./mqcreate1 /tmp.123
Result:Success
[gun@localhost PosixMsg]$ ls -l /dev/mqueue
-rw-------.  1 gun  gun    80 Янв 30 14:10 tmp.123
```

*Очередь уже существует*:

```
[gun@localhost PosixMsg]$ ./mqcreate1 -e /tmp.123
Result:File exists
```

*Свойства очереди*:

```
[gun@CentOS]$ cat /dev/mqueue/tmp.123
QSIZE:0       NOTIFY:0     SIGNO:0     NOTIFY_PID:0
```

Здесь:

- **QSIZE** – число байтов данных во всех сообщениях очереди;
- **NOTIFY** – метод уведомления (0 – **SIGEV_SIGNAL**; 1 – **SIGEV_NONE**; 2 – **SIGEV_THREAD**);
- **SIGNO** – номер сигнала, используемого для **SIGEV_SIGNAL**;
- **NOTIFY_PID** – если не 0, то это идентификатор (PID) процесса, использовавшего функцию **mq_notify**, чтобы зарегистрироваться на уведомление о появлении сообщения; предыдущие поля описывают настройки уведомления.

Эта версия программы названа mqcreate1, поскольку она будет улучшена после того, как будет описано использование атрибутов очереди. Разрешения на доступ к файлу очереди определяются константой **FILE_MODE** (чтение и запись для пользователя).

В той же папке сайта [6] имеется программа mqunlink.c, удаляющая из системы очередь сообщений. С ее помощью можно удалить очередь сообщений, созданную программой mqcreate1:

```
[gun@CentOS]$ ./mqunlink /tmp.123
Result=Success
[gun@CentOS]$ ls -l /dev/mqueue
итого 0
```

При этом будут удален файл из каталога **/dev/mqueue**, который относится к этой очереди.

### Функции mq_getattr и mq_setattr

У каждой очереди сообщений имеются четыре атрибута, которые могут быть получены функцией **mq_getattr** и установлены (по отдельности) функцией **mq_setattr**:

```
#include <mqueue.h>
int mq_getattr(mqd_t mqdes , struct mq_attr *attr );
int mq_setattr(mqd_t mqdes , const struct mq_attr
*attr , struct mq_attr *oattr );
```

Обе функции возвращают 0 в случае успешного завершения и –1 в случае возникновения ошибок.

Структура **mq_attr** хранит в себе эти четыре атрибута:

```
struct mq_attr {
long mq_flags; /* флаг очереди: 0, O_NONBLOCK */
long mq_maxmsg;/ *максимальное количество сообщений
в очереди */
long mq_msgsize; /* максимальный размер сообщения
(в байтах) */
long mq_curmsgs; // текущее количество сообщений в
очереди
}
```

Указатель на такую структуру может быть передан в качестве четвертого аргумента **mq_open**, что дает возможность установить пара-

метры **mq_maxmsg** и **mq_msgsize** в момент создания очереди. Другие два поля структуры функцией **mq_open** игнорируются.

Функция **mq_getattr** присваивает полям структуры, на которую указывает **attr**, текущие значения атрибутов очереди.

Функция **mq_setattr** устанавливает атрибуты очереди, но фактически используется только поле **mqflags** той структуры, на которую указывает **attr**, что дает возможность сбрасывать или устанавливать флаг запрета блокировки. Другие три поля структуры игнорируются: максимальное количество сообщений в очереди и максимальный размер сообщения могут быть установлены только в момент создания очереди, а количество сообщений в очереди можно только считать, но не изменить.

Кроме того, если указатель **oattr** ненулевой, возвращаются предыдущие значения атрибутов очереди (**mq_flags**, **mq_maxmsg**, **mq_msgsize**) и текущий статус очереди (**mq_curmsgs**).

Программа mqgetattr.c из указанного ранее раздела сайта открывает указанную очередь сообщений и выводит значения ее атрибутов.

Мы можем создать очередь сообщений и вывести значения ее атрибутов, устанавливаемые по умолчанию (программой mqcreate1.c):

```
[gun@CentOS]$ ./mqgetattr /tmp.123
Result:Success
max #msgs = 10, max #bytes/msg = 8192, #currently
on queue = 0
```

Теперь можно изменить программу mqcreate1.c таким образом (см. mqcreate2.c), чтобы при создании очереди иметь возможность указывать максимальное количество сообщений и максимальный размер сообщения. Заметим, что нужно обязательно задать оба параметра. Причем параметр командной строки, требующий аргумента, указывается с помощью двоеточия (после параметров **m** и **z**) в вызове **getopt**. В момент обработки символа параметр **optarg** указывает на аргумент. Если не указан ни один из двух новых параметров, то необходимо передать функции **mq_open** пустой указатель в качестве последнего аргумента. В противном случае передается указатель на структуру **attr**.

Запустим теперь новую версию программы в системе CentOS 6.9, указав параметры, превышающие значения по умолчанию: максимальное количество сообщений 20 и максимальный размер сообщения 16 384 байт.

```
[gun@CentOS]$ ./mqcreate2 -m 20 -z 16384 /tmp.123
Result:Invalid argument
```

Видим, что в данной реализации по умолчанию устанавливаются максимально возможные параметры очереди сообщений. Действительно, указав значения меньше предельных, получим:

```
[gun@CentOS]$ ./mqcreate2 -m 5 -z 4096 /tmp.123
Result:Success
[gun@CentOS]$ ./mqgetattr /tmp.123
max #msgs = 5, max #bytes/msg = 4096, #currently on
queue = 0
```

### Функции mqsend и mqreceive

Эти две функции предназначены для помещения сообщений в очередь и получения их оттуда. Каждое сообщение имеет свой приоритет, который представляет собой беззнаковое целое, не превышающее **MQ_PRIO_MAX**. Стандарт POSIX требует, чтобы эта константа была не меньше 32.

Функция **mq_receive** всегда возвращает старейшее в указанной очереди сообщение с максимальным приоритетом, и приоритет может быть получен вместе с содержимым сообщения и его длиной. Действие **mq_receive** отличается от действия **msgrcv** в System V (параграф 6.3.2). Сообщения System V имеют поле **type**, аналогичное по смыслу приоритету, но для функции **msgrcv** можно указать три различных алгоритма возвращения сообщений: старейшее сообщение в очереди, старейшее сообщение с указанным типом или старейшее сообщение с типом, не превышающим указанного значения.

```
#include <mqueue.h>
int mq_send(mqd_t mqdes , const char *ptr , size_t
len , unsigned int prio );
```

Функция возвращает 0 в случае успешного завершения или –1 в случае возникновения ошибок.

```
ssize_t mq_receive(mqd_t mqdes , char *ptr , size_t
len , unsigned int *priop );
```

Функция возвращает количество байтов в сообщении в случае успешного завершения или –1 в случае ошибки.

Первые три аргумента обеих функций аналогичны первым трем аргументам функций **write** и **read** соответственно.

Значение аргумента **len** функции **mq_receive** должно быть по крайней мере не меньше максимального размера сообщения, которое может быть помещено в очередь, то есть значения поля **mq_msgsize** структуры **mq_attr** для этой очереди. Если **len** оказывается меньше этой величины, немедленно возвращается ошибка **EMSGSIZE**. Это означает, что большинству приложений, использующих очереди сообщений POSIX, придется вызывать **mq_getattr** после открытия очереди для определения максимального размера сообщения, а затем выделять память под один или несколько буферов чтения этого размера. Требование, чтобы буфер был больше по размеру, чем максимально возможное сообщение, позволяет функции **mq_receive** не возвращать уведомление о том, что размер письма превышает объем буфера. Сравните это, например, с флагом **MSG_NOERROR** и ошибкой **E2BIG** для очередей сообщений System V (параграф 6.3.2) и флагом **MSG_TRUNC** для функции **recvmsg**, используемой с дейтаграммами UDP (глава 8).

Аргумент **prio** устанавливает приоритет сообщения для **mq_send**, его значение должно быть меньше **MQ_PRIO_MAX**. Если при вызове **mq_receive** аргумент **priop** является ненулевым указателем, то в нем сохраняется приоритет возвращаемого сообщения. Если приложению не требуется использование различных приоритетов сообщений, можно указывать его равным нулю для **mq_send** и передавать **mq_receive** нулевой указатель в качестве последнего аргумента.

Разрешена передача сообщений нулевой длины. В стандарте (POSIX.1) нигде не запрещена передача сообщений нулевой длины. Функция **mq_receive** возвращает количество байтов в сообщении (в случае успешного завершения работы) или –1 в случае возникновения ошибок, так что 0 обозначает сообщение нулевой длины.

Файл mqsend.c, доступный на сайте [6] в разделе «PosixMsg», содержит текст программы, помещающей сообщение в очередь.

И размер сообщения, и его приоритет являются обязательными аргументами командной строки. Буфер под сообщение выделяется функцией **calloc**, которая инициализирует его нулем.

Файл mqreceive.c, доступный на сайте в том же разделе «PosixMsg» [6], содержит текст программы, считывающей сообщение из очереди.

Параметр командной строки **-n** отключает блокировку. При этом программа возвращает сообщение об ошибке, если в очереди нет сообщений. Открыв очередь, необходимо получить ее атрибуты, вызвав **mq_getattr**. Обязательно нужно определить максимальный размер сообщения, потому что необходимо выделить буфер подходящего размера перед вызовом **mq_receive**. Программа выводит размер считываемого сообщения и его приоритет.

Воспользуемся этими и ранее упомянутыми программами, чтобы проиллюстрировать использование поля приоритета.

*1. Создаем очередь.*

```
[gun@CentOS]$ ./mqcreate2 /test1
Result:Success
```

*2. Смотрим на ее атрибуты.*

```
[gun@CentOS]$ ./mqgetattr /test1
max #msgs = 10, max #bytes/msg = 8192, #currently
on queue = 0
```

*3. Отправка 100 байт с некорректным значением приоритета.*

```
[gun@CentOS]$ ./mqsend /test1 100 99999
Sent:-1 bytes
```

*4. Отправка 100 байт, приоритет 6, возврат нуля – это успех.*

```
[gun@CentOS]$ ./mqsend /test1 100 6
Sent:0 bytes
```

*5. Отправка 50 байт, приоритет 18.*

```
[gun@CentOS]$./mqsend /test1 50 18
Sent:0 bytes
```

*6. Отправка 33 байт, приоритет 18.*

```
[gun@CentOS]$./mqsend /test1 33 18
Sent:0 bytes
```

*7. Проверка содержимого очереди.*

```
[gun@CentOS]$./mqgetattr /test1
max #msgs = 10, max #bytes/msg = 8192, #currently
on queue = 3
```

*8. Возвращается старейшее сообщение с наивысшим приоритетом.*

```
[gun@CentOS]$./mqreceive /test1
read 50 bytes, priority = 18
[gun@CentOS]$./mqreceive /test1
read 33 bytes, priority = 18
[gun@CentOS]$./mqreceive /test1
read 100 bytes, priority = 6
```

*9. Отключаем блокировку, убеждаемся, что сообщений больше нет.*

```
[gun@CentOS]$./mqreceive –n /test1
read -1 bytes, priority = 4029664
```

*10. Удаляем очередь.*

```
[gun@CentOS]$./mqunlink /test1
Result=Success
```

### Ограничения очередей сообщений

Ранее были описаны два ограничения, устанавливаемые для любой очереди в момент ее создания:

- **mq_maxmsg** – максимальное количество сообщений в очереди;
- **mq_msgsize** – максимальный размер сообщения.

Не существует каких-либо ограничений на эти значения, хотя в некоторых реализациях необходимо наличие в файловой системе места для файла требуемого размера. Кроме того, ограничения на эти величины могут накладываться реализацией виртуальной памяти.

Другие два ограничения определяются реализацией:

- **MQ_OPEN_MAX** – максимальное количество очередей сообщений, которые могут быть одновременно открыты каким-либо процессом (POSIX требует, чтобы эта величина была не меньше 8);
- **MQ_PRIO_MAX** – максимальное значение приоритета плюс один (POSIX требует, чтобы эта величина была не меньше 32).

Эти две константы часто определяются в заголовочном файле <unistd.h> и могут быть получены во время выполнения программы вызовом функции sysconf, как показано в тексте программы mqsysconf.c, доступной на сайте [6].

Запустив эту программу, получим:

```
[gun@CentOS]$ ./mqsysconf
MQ_OPEN_MAX = 256, MQ_PRIO_MAX = 32768
```

Можно также использовать утилиту **sysctl**:

```
[gun@CentOS]# sysctl -A | grep fs.mqueue
fs.mqueue.queues_max = 256
fs.mqueue.msg_max = 10
fs.mqueue.msgsize_max = 8192
fs.mqueue.msg_default = 10
fs.mqueue.msgsize_default = 8192
```

### Функция mq_notify

Один из недостатков очередей сообщений System V заключается в невозможности уведомить процесс о том, что в очередь было помещено сообщение. Можно заблокировать процесс при вызове **msgrcv**, но тогда невозможно выполнять другие действия во время ожидания сообщения. Если указать флаг отключения блокировки при вызове **msgrcv** (**IPC_NOWAIT**), процесс не будет заблокирован, но придется регулярно вызывать эту функцию, чтобы получить сообщение, когда оно будет отправлено. Такая процедура называется опросом, и на нее тратится лишнее время. Нужно, чтобы система сама уведомляла процесс о том, что в пустую очередь было помещено новое сообщение.

Очереди сообщений POSIX допускают асинхронное уведомление о событии, когда сообщение помещается в очередь. Это уведомление может быть реализовано либо отправкой сигнала, либо созданием программного потока (см. главу 7) для выполнения указанной функции.

Режим уведомления включается с помощью функции **mq_notify**:

```
#include <mqueue.h>
int mq_notify(mqd_t mqdes, const struct sigevent
*notification);
```

Функция возвращает 0 в случае успешного выполнения или –1 в случае ошибки. Она включает и выключает асинхронное уведомление о событии для указанной очереди. Структура **sigevent** впервые появилась в стандарте POSIX.1 для сигналов реального времени, которые в данном пособии не рассматриваются. Эта структура и все новые кон-

станты, относящиеся к сигналам, определены в заголовочном файле <signal.h>:

```
union sigval {
int sival_int; /* целое значение */
void *sival_ptr; /* указатель */
};
struct sigevent {
int sigev_notify; /* SIGEV_{NONE,SIGNAL,THREAD} */
int sigev_signo; /* номер сигнала, если
SIGEV_SIGNAL */
union sigval sigev_value; /* передается обработчику
сигнала или потоку */
/* Следующие два поля определены для SIGEV_THREAD */
void (*sigev_notify_function) (union sigval);
pthread_attr_t *sigev_notify_attributes;
```

Есть несколько правил, действующих для этой функции.

1. Если аргумент **notification** ненулевой, процесс ожидает уведомления при поступлении нового сообщения в указанную очередь, пустую на момент его поступления. Мы говорим, что процесс регистрируется на уведомление для данной очереди.

2. Если аргумент **notification** представляет собой нулевой указатель и процесс уже зарегистрирован на уведомление для данной очереди, то уведомление для него отключается.

3. Только один процесс может быть зарегистрирован на уведомление для любой данной очереди в любой момент.

4. При помещении сообщения в пустую очередь, для которой имеется зарегистрированный на уведомление процесс, оно будет отправлено только в том случае, если нет заблокированных в вызове **mq_receive** процессов для этой очереди. Таким образом, блокировка в вызове **mq_receive** имеет приоритет перед любой регистрацией на уведомление.

5. При отправке уведомления зарегистрированному процессу регистрация снимается. Процесс должен зарегистрироваться снова (если в этом есть необходимость), вызвав **mq_notify** еще раз.

Пример простейшей программы, включающей отправку сигнала **SIGUSR1** при помещении сообщения в пустую очередь, представлен в файле mqnotify.c на сайте [6]. В ней объявляено несколько глобальных переменных, используемых совместно функцией **main** и обработчи-

ком сигнала (**sig_usr1**). Программа открывает очередь сообщений, получает ее атрибуты и выделяет буфер считывания соответствующего размера.

Затем устанавливается обработчик для сигнала **SIGUSR1**. Для этого полю **sigev_notify** структуры **sigevent** присваивается значение **SIGEV_SIGNAL**, что говорит системе о необходимости отправки сигнала, когда очередь из пустой становится непустой. Полю **sigev_signo** присваивается значение, соответствующее тому сигналу, который необходимо получить. Затем вызывается функция **mq_notify**.

Функция **main** после этого зацикливается, и процесс приостанавливается при вызове функции **pause**, возвращающей –1 при получении сигнала.

Обработчик сигнала вызывает **mq_notify** для перерегистрации, считывает сообщение и выводит его длину. В этой программе игнорируется приоритет полученного сообщения.

Запустим эту программу в одном из окон:

```
[gun@CentOS]$ ./mqcreate /test1
[gun@CentOS]$ ./mqnotify /test1
```

Затем выполним следующую команду в другом окне:

```
[gun@CentOS]$ ./mqsend /test1 50 16
```

Как и ожидалось, программа mqnotifysig.c выведет сообщение:

```
SIGUSR1 received, read 50 bytes
```

Заметим, что пустые сообщения тоже вызывают отправку сигнала:

```
[gun@CentOS]$ ./mqnotify /test1
[gun@CentOS]$ ./mqsend /test1 0 16
SIGUSR1 received, read 0 bytes
```

Можно проверить, что только один процесс может быть зарегистрирован на получение уведомления в любой момент, запустив копию программы в другом окне:

```
[gun@CentOS]$ ./mqnotifysig /test1
Result:Device or resource busy
```

Это сообщение соответствует коду ошибки **EBUSY**.

Недостаток программы mqnotifysig.c в том, что она вызывает **mq_notify**, **mq_receive** и **printf** из обработчика сигнала. Ни одну из этих функций вызывать оттуда не следует. Функции, которые могут быть вызваны из обработчика сигнала, относятся к группе, называемой, согласно POSIX, async-signal-safe functions (функции, обеспечивающие безопасную обработку асинхронных сигналов).

Функции, которых нет в этом списке, не должны вызываться из обработчика сигнала. В списке отсутствуют стандартные функции библиотеки ввода-вывода и функции **pthread_XXX** для работы с потоками (см. главу 7). Из всех функций IPC, рассматриваемых в этом пособии, в список попали только **sem_post**, **read** и **write** (подразумевается, что последние две используются с программными каналами и FIFO).

Одним из способов исключения вызова каких-либо функций из обработчика сигнала является установка этим обработчиком глобального флага, который проверяется программой для получения информации о приходе сообщения. Проблема тут в том, что уведомление отсылается только в случае, когда сообщение помещается в пустую очередь. Если в очередь поступают два сообщения, прежде чем первое будет считано, то отсылается только одно уведомление.

### 6.4.3. Семафоры POSIX IPC

Семафоры POSIX (в отличие от семафоров System V IPC) не обязательно должны обрабатываться ядром. Их особенностью является наличие имен, которые могут соответствовать именам реальных файлов в файловой системе. На рис. 6.17 изображена схема, иллюстрирующая именованный семафор POSIX.

Заметим, что хотя именованные семафоры POSIX обладают именами в файловой системе, они не обязательно должны храниться в файлах. Во встроенной системе реального времени значение семафора, скорее всего, будет размещаться в ядре, а имя файла будет использоваться исключительно для идентификации семафора. При реализации с помощью отображения файлов в память значение семафора будет действительно храниться в файле, который будет отображаться в адресное пространство всех процессов, использующих семафор.

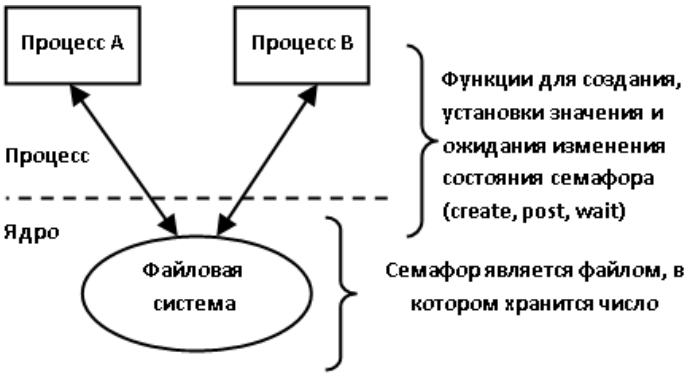



*Рис. 6.17.* Два процесса, использующие именованный семафор POSIX

На рис. 6.17 указаны три операции, которые могут быть применены к семафорам.

1. Создание семафора (*create*). При этом вызвавший процесс должен указать начальное значение (положительное число, часто 1, но может быть и 0).

2. Ожидание изменения значения семафора (*wait*). При этом производится проверка его значения и процесс блокируется, если значение оказывается меньше либо равным 0, а при превышении 0 значение уменьшается на 1. Основным требованием является атомарность выполнения операций проверки значения и последующего уменьшения значения семафора (как одной операции) по отношению к другим потокам. Это одна из причин, по которой семафоры System V были реализованы в середине 80-х как часть ядра. Операции с ними выполняются с помощью системных вызовов, что легко гарантирует их атомарность по отношению к другим процессам.

3. Установка значения семафора (*post*). Значение семафора увеличивается одной командой, и если в системе имеются процессы, ожидающие изменения значения семафора до величины, превосходящей 0, то один из них может быть пробужден. Как и операция ожидания, операция установки значения семафора также должна быть атомарной по отношению к другим процессам, работающим с этим семафором.

Стандартом POSIX описано два типа семафоров: именованные (*named*) и размещаемые в памяти (*memory-based* или *unnamed*). Именованный семафор POSIX был изображен на рис. 6.17. Неименованный,

или размещаемый в памяти (в том числе разделяемой) семафор, который можно использовать и для синхронизации потоков одного процесса (см. главу 7), изображен на рис. 6.18. В случае использования разделяемой памяти общий ее сегмент принадлежит адресному пространству нескольких процессов.

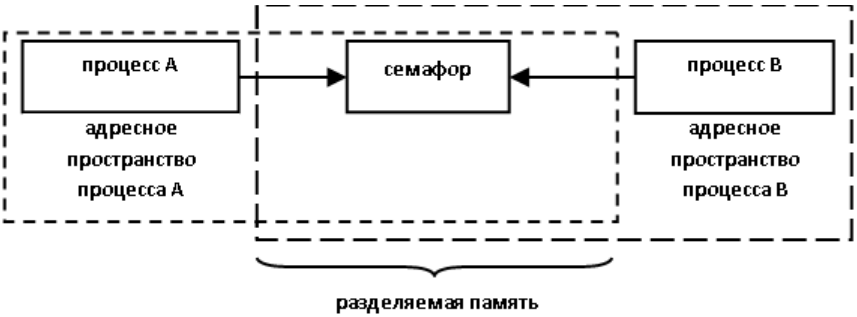



*Рис. 6.18.* Семафор, размещенный в разделяемой двумя процессами памяти

Далее сначала рассматриваются именованные семафоры POSIX, а затем – размещаемые в памяти.

### Функции sem_open, sem_close и sem_unlink

Функция **sem_open** создает новый именованный семафор или открывает существующий. Именованный семафор может использоваться для синхронизации выполнения потоков и процессов:

```
#include <semaphore.h>
sem_t *sem_open(char *name, int oflag, … /* mode_t
mode, unsigned int value  */);
```

Функция возвращает указатель на семафор в случае успешного завершения или **SEM_FAILED** в случае ошибки.

Аргумент **oflag** может принимать значения 0, **O_CREAT**, **O_CREAT | O_EXCL**, как описано ранее. Если указано значение **O_CREAT**, третий и четвертый аргументы функции являются обязательными. Аргумент **mode** указывает биты разрешений доступа (см. табл. 6.11), а **value** указывает начальное значение семафора. Это значение не может превышать константу **SEM_VALUE_MAX**, которая, согласно POSIX, должна быть не менее 32 767. Бинарные семафоры

обычно устанавливаются в 1, тогда как семафоры-счетчики чаще инициализируются бóльшими величинами.

При указании флага **O_CREAT** (без **O_EXCL**) семафор инициализируется только в том случае, если он еще не существует. Если семафор существует, ошибки не возникнет. Ошибка будет возвращена только в том случае, если указаны флаги **O_CREAT | O_EXCL**.

Возвращаемое значение представляет собой указатель на тип **sem_t**. Этот указатель впоследствии передается в качестве аргумента функциям **sem_close**, **sem_wait**, **sem_trywait**, **sem_post** и **sem_getvalue**.

Открыв семафор с помощью **sem_open**, можно потом закрыть его, вызвав **sem_close**:

```
#include <semaphore.h>
int sem_close(sem_t *sem );
```

Функция возвращает 0 в случае успешного завершения или –1 в случае ошибки.

Операция закрытия выполняется автоматически при завершении процесса для всех семафоров, которые были им открыты. Автоматическое закрытие осуществляется как при добровольном завершении работы (вызове **exit**), так и при принудительном (с помощью сигнала).

Закрытие семафора не удаляет его из системы. Именованные семафоры POSIX обладают по меньшей мере живучестью ядра. Значение семафора сохраняется, даже если ни один процесс не держит его открытым.

Именованный семафор удаляется из системы вызовом **sem_unlink**:

```
#include <semaphore.h>
int sem_unlink(const char *name );
```

Функция возвращает 0 в случае успешного завершения или –1 в случае ошибки.

Для каждого семафора ведется подсчет процессов, в которых он является открытым, и функция **sem_unlink** действует аналогично **unlink** для файлов. Таким образом, объект **name** может быть удален из файловой системы, даже если он открыт какими-либо процессами, но реальное удаление семафора не будет осуществлено до тех пор, пока он не будет окончательно закрыт.

### Функции sem_wait и sem_trywait

Функция **sem_wait** проверяет значение заданного семафора на положительность, уменьшает его на единицу и немедленно возвращает управление процессу. Если значение семафора при вызове функции равно нулю, процесс приостанавливается до тех пор, пока оно снова не станет больше нуля, после чего значение семафора будет уменьшено на единицу и произойдет возврат из функции. Ранее мы отметили, что операция «проверка и уменьшение» должна быть атомарной по отношению к другим потокам, работающим с этим семафором:

```
#include <semaphore.h>
int sem_wait(sem_t *sem );
int sem_trywait(sem_t *sem );
```

Обе функции возвращают 0 в случае успешного завершения или –1 в случае ошибки.

Разница между **sem_wait** и **sem_trywait** заключается в том, что последняя не приостанавливает выполнение процесса, если значение семафора равно нулю, а просто немедленно возвращает ошибку **EAGAIN**.

Возврат из функции **sem_wait** может произойти преждевременно, если будет получен сигнал. При этом возвращается ошибка с кодом **EINTR**.

### Функции sem_post и sem_getvalue

После завершения работы с семафором процесс (поток) вызывает **sem_post**. Этот вызов увеличивает значение семафора на единицу и возобновляет выполнение любых потоков, ожидающих изменения значения семафора:

```
#include <semaphore.h>
int sem_post(sem_t *sem );
int sem_getvalue(sem_t *sem , int *valp );
```

Обе функции возвращают 0 в случае успешного завершения или –1 в случае ошибки.

Функция **sem_getvalue** возвращает текущее значение семафора, помещая его в целочисленную переменную, на которую указывает **valp**. Если семафор заблокирован, возвращается либо 0, либо отрицательное число, модуль которого соответствует количеству потоков, ожидающих разблокирования семафора.

### Простые примеры

В файле semcreate.c, доступном на сайте [6] в разделе «PosixSem», приведен текст программы, создающей именованный семафор. При вызове программы можно указать параметр **-e**, обеспечивающий исключающее создание (если семафор уже существует, будет выведено сообщение об ошибке), а параметр **-i** с числовым аргументом позволяет задать начальное значение семафора, отличное от 1. В Linux, именованные семафоры создаются в виртуальной файловой системе, обычно монтируемой в **/dev/shm**, с именами вида **sem.имя** (по этой причине длина имени семафора ограничена **NAME_MAX-4**, а не **NAME_MAX** символами).

Программа в файле semunlink.c удаляет именованный семафор.

В файле semgetvalue.c приведен текст простейшей программы, которая открывает указанный именованный семафор, получает его текущее значение и выводит его.

Программа в файле semwait.c открывает именованный семафор, вызывает **sem_wait** (которая приостанавливает выполнение процесса, если значение семафора меньше либо равно 0, а при положительном значении семафора уменьшает его на 1), получает и выводит значение семафора, а затем останавливает свою работу навсегда при вызове **pause**.

В файле sempost.c приведена программа, которая выполняет операцию *post* для указанного семафора (то есть увеличивает его значение на 1), а затем получает значение этого семафора и выводит его.

Далее рассмотрим работу этих примеров.

Создадим именованный семафор в CentOS 6.9 и выведем его значение, устанавливаемое по умолчанию при инициализации:

```
[gun@CentOS]$ ./semcreate /test1
[gun@CentOS}$ ls -l /dev/shm/sem.test1
-rw-------. 1 gun gun 16 Фев  2 11:50
/dev/shm/sem.test1
[gun@CentOS]$ ./semgetvalue test1
value = 1
```

Теперь подождем изменения семафора и прервем работу программы, установившей блокировку:

```
[gun@CentOS]$ ./semwait /test1
pid 19830 has semaphore, value = 0
```

```
^C //клавиша прерывания работы
[gun@CentOS]$ ./semgetvalue test1
value = 0 //значение остается нулевым
```

Приведенный пример иллюстрирует упомянутые ранее особенности. Во-первых, значение семафора обладает живучестью ядра. Значение 1, установленное при создании семафора, хранится в ядре даже тогда, когда ни одна программа не пользуется этим семафором. Во-вторых, при выходе из программы semwait, заблокировавшей семафор, значение его не изменяется, то есть ресурс остается заблокированным. Это отличает семафоры от блокировок **fcntl**, которые снимались автоматически при завершении работы процесса.

Покажем теперь, что в этой реализации ОС нулевое значение семафора используется для блокирования семафора:

```
[gun@CentOS]$ ./semgetvalue test1
value = 0 // это значение сохранилось с конца предыдущего примера
[gun@CentOS]$ ./semwait /test1 & // запуск в фоновом режиме
[1] 20298
[gun@CentOS]$ ./semgetvalue test1
value = 0
[gun@CentOS]$ ./semwait /test1 & // запуск в фоновом режиме
[2] 20335
[gun@CentOS]$ ./semgetvalue test1
value = 0
[gun@ CentOS]$ ./sempost /test1
pid 20298 has semaphore, value = 0 // вывод программы semwait
value = 0 // значение после возвращения из sem_post
 [gun@CentOS]$ ./sempost /test1
value = 1 // значение после возвращения из sem_post
pid 20335 has semaphore, value = 0 // вывод программы semwait
[gun@CentOS]$ ./semgetvalue /test1
value = 1
[gun@CentOS]$ ./sempost /test1
```

```
value = 2 // значение после возвращения из sem_post
[gun@CentOS]$ ./semgetvalue /test1
value = 2
```

Именованные семафоры POSIX могут быть размещены не только в виртуальной файловой системе, но и в отображаемом на память файле. Отображаемые в память файлы описаны в следующем разделе.

По окончании работы с примерами не забудем удалить семафор:

```
[gun@CentOS]$ ./semunlink /test1
Result:Success
```

Убедимся визуально, что семафор удален:

```
[gun@CentOS]$ ls -l /dev/shm/sem.test1
ls: невозможно получить доступ к
/dev/shm/sem.test1: Нет такого файла или каталога
```

### Функции sem_init и sem_destroy

До сих пор обсуждались только именованные семафоры POSIX. Они идентифицируются аргументом **name**, обычно представляющим собой имя файла в файловой системе. Стандарт POSIX описывает также семафоры, размещаемые в памяти, память под которые (типа **sem_t**) выделяет приложение, а инициализируются они системным вызовом:

```
#include <semaphore.h>
int sem_init(sem_t *sem,  int shared,  unsigned int
value );
```

Функция возвращает 0 в случае успешного завершения или –1 в случае ошибки.

Размещаемый в памяти семафор инициализируется вызовом **sem_init**. Аргумент **sem** указывает на переменную типа **sem_t**, место под которую должно быть выделено приложением. Если аргумент **shared** равен 0, то семафор используется потоками одного процесса (см. главу 7), в противном случае доступ к нему могут иметь несколько процессов. Если аргумент **shared** ненулевой, семафор должен быть размещен в одном из видов разделяемой памяти и должен быть доступен всем процессам, использующим его. Как и в вызове **sem_open**, аргумент **value** задает начальное значение семафора.

Обратите внимание, что для размещаемого в памяти семафора нет ничего аналогичного флагу **O_CREAT**: функция **sem_init** всегда инициализирует значение семафора. Следовательно, нужно быть внимательным, чтобы вызывать **sem_init** только один раз для каждого семафора. При вызове **sem_init** для уже инициализированного семафора результат непредсказуем.

После завершения работы с размещаемым в памяти семафором его можно уничтожить, вызвав **sem_destroy**:

```
int sem_destroy(sem_t *sem );
```

Функция возвращает 0 в случае успешного завершения или –1 в случае ошибки.

Размещаемый в памяти семафор может быть использован в тех случаях, когда нет необходимости использовать имя, связываемое с именованным семафором. Именованные семафоры обычно используются для синхронизации работы неродственных процессов. Имя в этом случае используется для идентификации семафора.

Размещаемый в памяти семафор не утрачивает функциональности до тех пор, пока память, в которой он размещен, еще доступна какому-либо процессу.

Если размещаемый в памяти семафор совместно используется потоками одного процесса (аргумент **shared** при вызове **sem_init** равен 0), семафор обладает живучестью процесса и удаляется при завершении последнего.

Если размещаемый в памяти семафор совместно используется несколькими процессами (аргумент **shared** при вызове **sem_init** равен 1), он должен располагаться в разделяемой памяти, и в этом случае семафор существует столько, сколько существует эта область памяти, поскольку и разделяемая память POSIX, и разделяемая память System V обладают живучестью ядра.

### Ограничения на семафоры

Стандартом POSIX определены два ограничения на семафоры:

- **SEM_NSEMS_MAX** – максимальное количество одновременно открытых семафоров для одного процесса (POSIX требует, чтобы это значение было не менее 256);
- **SEM_VALUE_MAX** – максимальное значение семафора (POSIX требует, чтобы оно было не меньше 32 767).

Две эти константы обычно определены в заголовочном файле <unistd.h> и могут быть получены во время выполнения вызовом **sysconf**, как показано в файле semsysconf.c, доступном на сайте [6].

При запуске этой программы в системе CentOS 6.9 получим следующий результат:

```
[gun@CentOS]$ ./semsysconf
SEM_NSEMS_MAX = 256, SEM_VALUE_MAX = 32767
```

### 6.4.4. Разделяемая память POSIX IPC

Стандарт POSIX.1 предоставляет два механизма совместного использования областей памяти для неродственных процессов:

1. Отображение файлов в память: файл открывается вызовом **open**, а его дескриптор используется при вызове функции **mmap** для отображения содержимого файла в адресное пространство процесса. Этот метод позволяет реализовать совместное использование памяти как для родственных, так и для неродственных процессов.

2. Объекты разделяемой памяти: функция **shm_open** открывает объект IPC с именем стандарта POSIX (например, полным именем объекта файловой системы), возвращая дескриптор, который может быть использован для отображения в адресное пространство процесса вызовом **mmap**.

Оба метода требуют вызова **mmap**. Отличие состоит в методе получения дескриптора, являющегося аргументом **mmap**: в первом случае он возвращается функцией **open**, а во втором – **shm_open**. Это иллюстрирует рис. 6.19. Стандарт POSIX называет объектами памяти (memory objects) и отображенные в память файлы, и объекты разделяемой памяти стандарта POSIX.

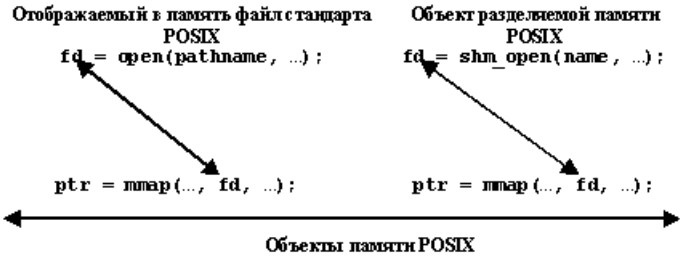



*Рис. 6.19.* Объекты памяти POSIX: отображаемые в память файлы и объекты разделяемой памяти

Поскольку оба подхода требуют использования функций для работы с отображаемыми на память файлами, вначале изучим их.

### Функции mmap, munmap и msync

Функция **mmap** отображает в адресное пространство процесса файл или объект разделяемой памяти POSIX. Мы используем эту функцию в следующих ситуациях:

1. С обычными файлами для обеспечения ввода-вывода через отображение в память.
2. Со специальными файлами для обеспечения неименованного отображения памяти (несуществующие файлы, как в Windows API и файл **/dev/zero**), но такие файлы поддерживаются только в ОС семейства BSD.
3. С функцией **shm_open** для создания участка разделяемой неродственными процессами памяти POSIX.

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot,
int flags, int fd, off_t offset );
```

Функция возвращает начальный адрес участка памяти в случае успешного завершения или **MAP_FAILED** в случае ошибки.

Аргумент **addr** может указывать начальный адрес участка памяти процесса, в который следует отобразить содержимое дескриптора **fd**. Обычно ему присваивается значение нулевого указателя, что говорит ядру о необходимости выбрать начальный адрес самостоятельно. В любом случае функция возвращает начальный адрес сегмента памяти, выделенной для отображения.

Аргумент **len** задает длину отображаемого участка в байтах; участок может начинаться не с начала файла, а с некоторого места, задаваемого аргументом **offset**. Обычно **offset** = 0. На рис. 6.20 изображена схема отображения объекта в память.

Защита участка памяти с отображенным объектом обеспечивается с помощью аргумента **prot** и констант, приведенных в табл. 6.13. Обычное значение этого аргумента – **PROT_READ | PROT_WRITE**, что обеспечивает доступ на чтение и запись.

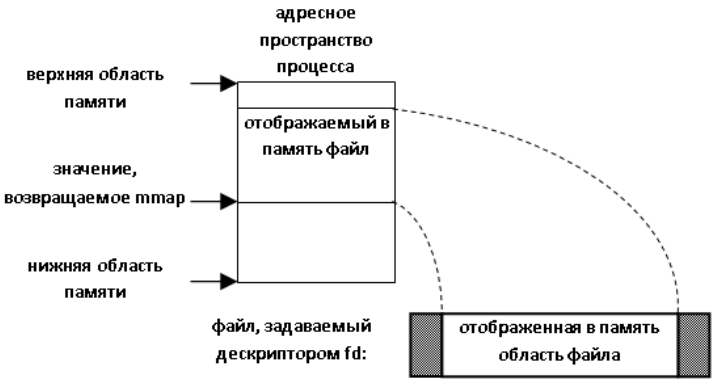



*Рис. 6.20.* Пример отображения файла в память

Таблица 6.13

**Аргумент `prot` для вызова `mmap`**

| `prot` | Описание |
|---|---|
| **`PROT_READ`** | Данные могут быть считаны |
| **`PROT_WRITE`** | Данные могут быть записаны |
| **`PROT_EXEC`** | Данные могут быть выполнены |
| **`PROT_NONE`** | Доступ к данным закрыт |

Аргумент **`flags`** может принимать значения из табл. 6.14. Можно указать только один из флагов – **`MAP_SHARED`** или **`MAP_PRIVATE`**, прибавив к нему при необходимости **`MAP_FIXED`**. Если указан флаг **`MAP_PRIVATE`**, все изменения будут производиться только с образом объекта в адресном пространстве процесса, другим процессам они доступны не будут. Если же указан флаг **`MAP_SHARED`**, изменения отображаемых данных видны всем процессам, совместно использующим объект.

Для обеспечения переносимости программ флаг **`MAP_FIXED`** указывать не следует. Если он не указан, но аргумент **`addr`** представляет собой ненулевой указатель, интерпретация этого аргумента зависит от реализации. Ненулевое значение **`addr`** обычно трактуется как указатель на желаемую область памяти, в которую нужно произвести отображение. В переносимой программе значение **`addr`** должно быть нулевым и флаг **`MAP_FIXED`** не должен быть указан.

Таблица 6.14

**Аргумент `flag` для вызова `mmap`**

| `flag` | Описание |
|---|---|
| **`MAP_SHARED`** | Изменения передаются другим процессам |
| **`MAP_PRIVATE`** | Изменения не передаются другим процессам и не влияют на отображенный объект |
| **`MAP_FIXED`** | Аргумент **`addr`** интерпретируется как адрес памяти |

Одним из способов добиться совместного использования памяти родительским и дочерним процессами является вызов **`mmap`** с флагом **`MAP_SHARED`** перед вызовом **`fork`**. Стандарт POSIX.1 гарантирует в этом случае, что все отображения памяти, установленные родительским процессом, будут унаследованы дочерним. Более того, изменения в содержимом объекта, вносимые родительским процессом, будут видны дочернему процессу, и наоборот.

Для отключения отображения объекта в адресное пространство процесса используется вызов **`munmap`**:

```
#include <sys/mman.h>
int munmap(void *addr,  size_t len );
```

Функция возвращает 0 в случае успешного завершения или –1 в случае ошибки.

Аргумент **`addr`** должен содержать адрес, возвращенный **`mmap`**, а **`len`** – длину области отображения. После вызова **`munmap`** любые попытки обратиться к этой области памяти приведут к отправке процессу сигнала **`SIGSEGV`** (предполагается, что эта область памяти не будет снова отображена вызовом **`mmap`**).

Если область была отображена с флагом **`MAP_PRIVATE`**, все внесенные за время работы процесса изменения сбрасываются.

В изображенной на рис. 6.20 схеме ядро обеспечивает синхронизацию содержимого файла, отображенного в память, с самой памятью при помощи алгоритма работы с виртуальной памятью (если сегмент был отображен с флагом **`MAP_SHARED`**). Если происходит изменение содержимого ячейки памяти, в которую отображен файл, через некоторое время содержимое файла будет соответствующим образом изменено ядром. Однако в некоторых случаях необходимо, чтобы содер-

жимое файла всегда было в соответствии с содержимым памяти. Тогда для осуществления моментальной синхронизации вызывают функцию **msync**:

```
#include <sys/mman.h>
int msync(void *addr,  size_t len , int flags );
```

Функция возвращает 0 в случае успешного завершения или –1 в случае ошибки.

Аргумент **flags** представляет собой комбинацию констант из табл. 6.15.

Таблица 6.15

**Значения аргумента flags для функции msync**

| Константа | Описание |
|---|---|
| **MS_ASYNC** | Осуществлять асинхронную запись |
| **MS_SYNC** | Осуществлять синхронную запись |
| **MS_INVALIDATE** | Сбросить кэш |

Из двух констант **MS_ASYNC** и **MS_SYNC** указать нужно одну и только одну. Отличие между ними в том, что возврат из функции при указании флага **MS_ASYNC** происходит сразу же, как только данные для записи будут помещены в очередь ядром, а при указании флага **MS_SYNC** возврат происходит только после завершения операций записи. Если указан и флаг **MS_INVALIDATE**, все копии файла, содержимое которых не совпадает с его текущим содержимым, считаются устаревшими. Последующие обращения к этим копиям приведут к считыванию данных из файла.

Удобство работы с отображением в память содержимого файла состоит в том, что все операции ввода-вывода осуществляются ядром и скрыты от программиста, а он просто пишет код, считывающий и записывающий данные в некоторую область памяти. Ему не приходится вызывать **read**, **write** или **lseek**. Часто это заметно упрощает код.

Следует, однако, иметь в виду, что не все файлы могут быть отображены в память. Попытка отобразить дескриптор, указывающий на терминал или сокет, приведет к возвращению ошибки при вызове **mmap**. К дескрипторам этих типов доступ осуществляется только с помощью **read** и **write** (и аналогичных вызовов).

Другой целью использования **mmap** может являться разделение памяти между неродственными процессами. В этом случае содержимое файла становится начальным содержимым разделяемой памяти и любые изменения, вносимые в нее процессами, копируются обратно в файл (что дает этому виду IPC живучесть файловой системы). Предполагается, что при вызове **mmap** указывается флаг **MAP_SHARED**, необходимый для разделения памяти между процессами.

### Функции shm_open и shm_unlink

Процесс получения доступа к объекту разделяемой памяти POSIX выполняется в два этапа:

1. Вызов **shm_open** с именем IPC в качестве аргумента позволяет либо создать новый объект разделяемой памяти, либо открыть существующий.

2. Вызов **mmap** позволяет отобразить разделяемую память в адресное пространство вызвавшего процесса.

Аргумент **name**, указанный при первом вызове **shm_open**, должен впоследствии использоваться всеми прочими процессами, желающими получить доступ к данной области памяти.

Причина, почему этот процесс выполняется в два этапа вместо одного, на котором в ответ на имя объекта возвращался бы адрес соответствующей области памяти, заключается в том, что функция **mmap** уже существовала, когда эта форма разделяемой памяти была включена в стандарт POSIX.

```
#include <sys/mman.h>
int shm_open(const char *name,  int oflag,  mode_t
mode );
```

Функция возвращает неотрицательный дескриптор в случае успешного завершения или –1 в случае ошибки.

```
int shm_unlink(const char *name );
```

Функция возвращает 0 в случае успешного завершения или –1 в случае ошибки.

Аргумент **oflag** должен содержать флаг **O_RDONLY** либо **O_RDWR** и один из следующих: **O_CREAT**, **O_EXCL**, **O_TRUNC**. Флаги **O_CREAT** и **O_EXCL** были описаны ранее. Если вместе с флагом **O_RDWR** указан

флаг **O_TRUNC**, существующий объект разделяемой памяти будет укорочен до нулевой длины.

Аргумент **mode** задает биты разрешений доступа (табл. 6.11) и используется только при указании флага **O_CREAT**. Обратите внимание, что в отличие от функций **mq_open** и **sem_open** для **shm_open** аргумент **mode** указывается всегда. Если флаг **O_CREAT** не указан, значение аргумента **mode** может быть нулевым.

Возвращаемое значение **shm_open** представляет собой целочисленный дескриптор, который может использоваться при вызове **mmap** в качестве пятого аргумента.

Функция **shm_unlink** удаляет имя объекта разделяемой памяти. Как и другие подобные функции (удаление файла из файловой системы, удаление очереди сообщений и именованного семафора POSIX), она не выполняет никаких действий до тех пор, пока объект не будет закрыт всеми открывшими его процессами. Однако после вызова **shm_unlink** последующие вызовы **open**, **mq_open** и **sem_open** выполняться не будут.

### Функции ftruncate и fstat

Размер файла или объекта разделяемой памяти можно изменить вызовом **ftruncate**:

```
#include <unistd.h>
int ftruncate(int fd,  off_t length );
```

Функция возвращает 0 в случае успешного завершения или –1 в случае ошибки.

Стандарт POSIX делает некоторые различия в определении действия этой функции для обычных файлов и для объектов разделяемой памяти.

1. Для обычного файла: если размер файла превышает значение **length**, избыточные данные отбрасываются. Если размер файла оказывается меньше значения **length**, действие функции не определено. Поэтому для переносимости следует использовать следующий способ увеличения длины обычного файла: вызов **seek** со сдвигом **length-1** и запись одного байта в файл. К счастью, почти все реализации UNIX поддерживают увеличение размера файла вызовом **ftruncate**.

2. Для объекта разделяемой памяти **ftruncate** устанавливает размер объекта равным значению аргумента **length**.

Итак, **ftruncate** вызывается для установки размера только что созданного объекта разделяемой памяти или изменения размера существующего объекта. При открытии существующего объекта разделяемой памяти следует воспользоваться **fstat** для получения информации о нем:

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

Функция возвращает 0 в случае успешного завершения или –1 в случае ошибки.

В структуре **stat** содержится больше десятка полей, но только четыре из них содержат актуальную информацию, если **fd** представляет собой дескриптор области разделяемой памяти:

```
struct stat {
…
mode_t st_mode; /* mode: S_I{RW}{USR,GRP,OTH} */
uid_t st_uid; /* UID владельца */
gid_t st_gid; /* GID владельца */
off_t st_size; /* размер в байтах */
…
};
```

Пример использования этих двух функций приведен ниже.

### Простые программы

Приведем несколько примеров программ, работающих с разделяемой памятью POSIX.

Программа shmcreate.c, текст которой доступен на сайте [6] в разделе «PosixShm», создает объект разделяемой памяти с указанным именем и длиной. Вызов **shm_open** создает объект разделяемой памяти. Когда указан параметр **–e**, будет возвращена ошибка в том случае, если такой объект уже существует. Вызов **ftruncate** устанавливает длину (размер объекта), а **mmap** отображает его содержимое в адресное пространство процесса. Затем программа завершает работу. Поскольку

разделяемая память POSIX обладает живучестью ядра, объект разделяемой памяти при этом не исчезает.

В файле shmunlink.c приведен текст тривиальной программы, удаляющей имя объекта разделяемой памяти из системы.

В файле shmwrite.c приведен текст программы, записывающей последовательность 0, 1, 2, …, 254, 255, 0, 1 и т. д. в объект разделяемой памяти. Объект разделяемой памяти открывается вызовом **shm_open**. Его размер можно узнать с помощью **fstat**. Затем файл отображается в память вызовом **mmap**, после чего его дескриптор может быть закрыт.

Программа в файле shmread.c проверяет значения, помещенные в разделяемую память программой shmwrite. Объект разделяемой памяти открывается только для чтения, его размер получается вызовом **fstat**, после чего он отображается в память с доступом только на чтение, а дескриптор закрывается.

Далее рассмотрим работу этих примеров.

Создадим объект разделяемой памяти с именем temp объемом 123 456 байт в системе CentOS 6.9:

```
[gun@CentOS]$ ./shmcreate /temp 123456
Result:Success
[gun@CentOS]$ ls -l /dev/shm/temp
-rw-------. 1 gun gun 123456 Фев  3 18:13
/dev/shm/temp
[gun@CentOS]$ od /dev/shm/temp
0000000 000000 000000 000000 000000 000000 000000 000000
000000 000000
*
0361100
```

Здесь видно, что файл с указываемым при создании объекта разделяемой памяти именем появляется в виртуальной файловой системе, там же, где и семафоры. Соответственно, объекты можно просматривать стандартными утилитами **ls** и **cat**. Используя программу **od**, можно выяснить, что после создания файл целиком заполнен нулями (восьмеричное число 0361100 – сдвиг, соответствующий байту, следующему за последним байтом файла, – эквивалентно десятичному 123 456).

Запустим программу shmwrite и убедимся в правильности записываемых значений с помощью программы **od**:

```
[gun@CentOS]$ ./shmwrite /temp
[gun@CentOS]$ od -x /dev/shm/temp |head -3
0000000 0100 0302 0504 0706 0908 0b0a 0d0c 0f0e
0000020 1110 1312 1514 1716 1918 1b1a 1d1c 1f1e
0000040 2120 2322 2524 2726 2928 2b2a 2d2c 2f2e
```

Проверим содержимое разделяемой памяти с помощью программы **shmread**:

```
[gun@CentOS]$ ./shmread /temp
```

Ошибок не обнаружено. Удалим объект, запустив программу **shmunlink**:

```
[gun@CentOS]$ ./shmunlink /temp
Result:Success
```

Убедимся визуально, что объект удален:

```
[gun@CentOS]$ ls -l /dev/shm/temp
ls: невозможно получить доступ к /dev/shm/temp: Нет такого файла или каталога
```

### Ограничения на разделяемую память

Ограничения на разделяемую память можно определить, просматривая виртуальную файловую систему.

*Максимальное количество объектов*:

```
[gun@CentOS]# cat /proc/sys/kernel/shmmni
4096
```

*Максимальный размер сегмента*:

```
[gun@CentOS]# cat /proc/sys/kernel/shmmax
4294967295
[gun@CentOS]# cat /proc/sys/kernel/shmall
268435456
```

## Вопросы для самопроверки

**1. POSIX IPC.**

1.1. Как именуются и где размещаются объекты IPC POSIX?

1.2. Опишите константы, используемые при создании и открытии объектов IPC.

1.3. Опишите константы, определяющие права доступа при создании объектов IPC.

1.4. Опишите последовательность логических операций при создании/открытии объекта IPC.

1.5. Опишите порядок проверки прав доступа при открытии объектов IPC.

**2. Очереди сообщений POSIX IPC.**

2.1. Опишите отличия очереди сообщений стандарта POSIX от стандарта System V.

2.2. Какова структура сообщения в очереди сообщений стандарта POSIX?

2.3. Опишите функцию создания очереди сообщений POSIX.

2.4. Опишите особенности закрытия и удаления очереди сообщений POSIX.

2.5. Каковы особенности компиляции программ, использующих очереди сообщений POSIX?

2.6. Какие команды ОС применимы для работы с очередями сообщений POSIX?

2.7. Опишите атрибуты очереди сообщений POSIX.

2.8. Опишите функции установки и чтения атрибутов очереди сообщений POSIX.

2.9. Опишите функцию отправки сообщений в очередь сообщений POSIX.

2.10. Опишите функцию чтения сообщений из очереди сообщений POSIX.

2.11. Какие существуют системные ограничения на очереди сообщений POSIX?

2.12. Опишите назначение функции **`mq_notify`**.

2.13. Опишите правила применения функции **`mq_notify`**.

2.14. Опишите недостатки применения функции **`mq_notify`**.

**3. Семафоры POSIX IPC**

3.1. Что представляют собой именованные и размещаемые в памяти семафоры POSIX?

3.2. Опишите функцию создания/открытия именованного семафора POSIX.

3.3. Опишите особенности закрытия и удаления именованного семафора POSIX.

3.4. Опишите функции ожидания именованного семафора POSIX.

3.5. Опишите функции **`sem_post`** и **`sem_getvalue`**.

3.6. Опишите функции для работы с анонимными (размещаемыми в памяти) семафорами POSIX.

3.7. Опишите системные ограничения на семафоры POSIX.

**4. Разделяемая память POSIX IPC**

4.1. Опишите механизмы совместного использования разделяемой памяти для неродственных процессов.

4.2. Перечислите варианты использования функции **`mmap`**.

4.3. Перечислите параметры функции **`mmap`**.

4.4. Перечислите параметры защиты разделяемой памяти и флаги доступа к ней.

4.5. Опишите особенности отключения объекта разделяемой памяти.

4.6. Опишите функцию **`msync`** и ее параметры.

4.7. В чем заключаются преимущества и ограничения при отображении файла в память?

4.8. Опишите этапы получения доступа к объекту разделяемой памяти POSIX.

4.9. Опишите параметры функции **`shm_open`**.

4.10. Опишите особенности удаления объекта разделяемой памяти.

4.11. Опишите особенности применения функции **`ftruncate`** для файлов и объектов разделяемой памяти.

4.12. Опишите применение функции **`fstat`** для объектов разделяемой памяти.

4.13. Опишите системные ограничения на объекты разделяемой памяти POSIX.

## Упражнения

1. Выполните лабораторную работу № 5 из лабораторного практикума, используя очереди сообщений POSIX.

2. Выполните лабораторную работу № 6 из лабораторного практикума, используя разделяемую память и семафоры POSIX.

# Глава 7. Многопоточное программирование в Linux

## 7.1. Создание потоков и управление ими

### 7.1.1. Создание потоков

Хотя концепция процессов в системах UNIX применяется уже очень давно, возможность использовать несколько потоков внутри одного процесса появилась относительно недавно. Стандарт потоков POSIX.1, называемый *Pthreads*, был принят в 1995 году. С точки зрения взаимодействия процессов все потоки одного процесса имеют общие глобальные переменные (то есть поточной модели свойственно использование общей памяти).

Традиционная модель процессов в UNIX поддерживает только один поток управления на процесс. Концептуально это то же, что и модель, основанная на потоках, в случае, когда каждый процесс состоит из одного потока. При наличии поддержки Pthreads программа также запускается как процесс, состоящий из одного потока управления. Поведение такой программы ничем не отличается от поведения традиционного процесса, пока она не создаст дополнительные потоки управления. Создание дополнительных потоков производится с помощью функции **pthread_create**:

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const
pthread_attr_t *attr,void* (*start)(void *),
void *arg);
```

При создании нового потока нельзя заранее предположить, кто первым получит управление – вновь созданный поток или поток, вызвавший функцию **pthread_create**. Новый поток имеет доступ к

адресному пространству процесса и наследует от вызывающего потока среду окружения.

Упрощенно вызов **pthread_create(&thr,NULL,start,NULL)** создаст поток, который начнет выполнять функцию **start** и запишет в переменную **thr** идентификатор созданного потока.

Первый аргумент этой функции, **thr**, – это указатель на переменную типа **pthread_t**. В нее будет записан идентификатор созданного потока, который впоследствии можно будет передавать другим вызовам, когда потребуется сделать что-либо с этим потоком.

Здесь наблюдается особенность POSIX API, а именно непрозрачность базовых типов. Дело в том, что практически ничего нельзя сказать про тип **pthread_t**. Единственное, что сказано в стандарте, – что эти значения можно копировать и что, используя вызов **int pthread_equal(pthread_t thr1, pthread_t thr2)**, можно установить, что оба идентификатора, **thr1** и **thr2**, идентифицируют один и тот же поток (при этом они вполне могут быть неравны в смысле оператора равенства).

Второй аргумент функции **pthread_create, attr**, – указатель на переменную типа **pthread_attr_t**, которая задает набор свойств создаваемого потока. Это вторая особенность POSIX API, а именно концепция атрибутов. В этом API во всех случаях, когда при создании или инициализации некоторого объекта необходимо задать набор дополнительных его свойств, вместо указания этого набора при помощи параметров вызова функции создания объекта используется передача предварительно сконструированного набора атрибутов.

Такое решение имеет по крайней мере два преимущества. Во-первых, можно зафиксировать набор параметров функции без угрозы его изменения в дальнейшем, когда у этого объекта появятся новые свойства. Во-вторых, можно многократно использовать один и тот же набор атрибутов для создания множества объектов.

Третий аргумент вызова **pthread_create** – это указатель на функцию типа **void*()(void*)**. Именно эту функцию и начинает выполнять вновь созданный поток, при этом в качестве параметра этой функции передается четвертый аргумент вызова **pthread_create**. Таким образом, можно, с одной стороны, параметризовать создаваемый поток кодом, который он будет выполнять, с другой стороны – параметризовать его различными данными, передаваемыми коду.

Функция **pthread_create** возвращает нулевое значение в случае успеха и ненулевой код ошибки в случае неудачи. Это также одна из особенностей POSIX API, вместо стандартного для UNIX подхода, когда функция возвращает лишь некоторый индикатор ошибки, а код ошибки устанавливает в переменной **errno**. Функции POSIX API возвращают код ошибки в результате своего аргумента.

В качестве резюме рассмотрим пример thread1.c, который можно скачать с сайта [6] из папки «Threads».

Хотя функции работы с потоками описаны в файле включения **pthread.h**, на самом деле они находятся в библиотеке **libgcc.a**. Поэтому процесс компиляции и сборки многопоточной программы выполняется так:

```
gcc -Wall -o test test.c -lpthread
```

### 7.1.2. Завершение потоков

Поток завершается, когда происходит возврат из функции. Если нужно получить возвращаемое этой функцией значение, надо воспользоваться такой функцией:

```
int pthread_join(pthread_t thread, void**
value_ptr);
```

Эта функция блокирует текущий поток, дожидается завершения потока с идентификатором **thread** и записывает возвращаемое ею значение в переменную, на которую указывает **value_ptr**. При этом освобождаются все ресурсы, связанные с потоком, потому эта функция может быть вызвана для данного потока только один раз. Если поток был принудительно завершен, по адресу **value_ptr** будет записано значение **PTHREAD_CANCELED**.

Вызов функции **pthread_join** автоматически переводит поток в сигнальное состояние, которое позволяет освободить ресурсы потока. Если он уже находится в сигнальном состоянии, поток, вызвавший **pthread_join**, получит код ошибки **EINVAL**.

Если возврат значения через **pthread_join** неудобен, допустим, необходимо получить несколько значений или данные из нескольких потоков, то следует воспользоваться каким-либо другим механизмом – например организовать очередь возвращаемых значений или возвращать значение в структуре, указатель на которую передают в качестве

параметра потока. То есть использование **pthread_join** – это вопрос удобства, а не догма, в отличие от случая пары **fork() - wait()**.

В том случае, если нужно использовать другой механизм возврата или возвращаемое значение просто не интересует, то можно отсоединить (**detach**) поток, сказав тем самым, что нужно освободить ресурсы, связанные с потоком, сразу по завершении функции потока. Сделать это можно несколькими способами.

Во-первых, можно сразу создать поток отсоединенным, задав соответствующий объект атрибутов при вызове **pthread_create**.

Во-вторых, любой поток можно отсоединить, вызвав в любой момент его жизни (то есть после вызова **pthread_create** и до вызова **pthread_join**) функцию **int pthread_detach(pthread_t thread)** и указав ей в качестве параметра идентификатор потока. При этом (в-третьих), поток вполне может отсоединить сам себя, получив свой идентификатор при помощи функции **pthread_t pthread_self(void)**. Поскольку к моменту возврата из функции **pthread_create** порожденный поток уже может завершиться и его идентификатор будет освобожден, третий способ предпочтительнее второго. Следует подчеркнуть, что отсоединение потока никоим образом не влияет на процесс его выполнения, а просто помечает поток как готовый по завершении к освобождению ресурсов.

Под освобождаемыми ресурсами подразумеваются в первую очередь стек, память, в которую сохраняется контекст потока, данные, специфичные для потока, и т. п. Сюда не входят ресурсы, выделяемые явно, например память, выделяемая через **malloc**, или открываемые файлы. Подобные ресурсы следует освобождать явно.

Помимо возврата (**return**) из функции потока существует вызов, аналогичный вызову **exit()** для процессов:

```
int pthread_exit(void *value_ptr);
```

Этот вызов завершает выполняемый поток, например, при возникновении какой-либо ошибки, возвращая в качестве результата его выполнения **value_ptr**. При вызове этой функции поток из нее просто не возвращается. Нужно помнить, что функция **exit()** по-прежнему завершает процесс, то есть уничтожает все потоки.

Рассмотрим соответствующий пример, thread2.c.

Для досрочного завершения потока извне его можно воспользоваться функцией **pthread_cancel(pthread_t thread)**. Единственным ее

аргументом является идентификатор потока **thread**. Функция **pthread_cancel()** возвращает 0 в случае успеха и ненулевое значение в случае ошибки. Хотя **pthread_cancel()** может завершить поток досрочно, ее нельзя назвать средством принудительного завершения потоков (аналогом функции **kill()** для процессов), поскольку поток может не только самостоятельно выбрать порядок завершения, но и игнорировать этот вызов. Вызов функции **pthread_cancel()** следует рассматривать как запрос на выполнение досрочного завершения потока. Функция **pthread_setcancelstate()** определяет, будет ли поток реагировать на обращение к нему с помощью **pthread_cancel()**, или не будет. У функции **pthread_setcancelstate()** два параметра: параметр **state** типа **int** и параметр **oldstate** типа ***int**. В первом передается значение, указывающее, как поток должен реагировать на запрос **pthread_cancel()**, а в переменную, чей адрес был передан во втором параметре, функция записывает прежнее значение. Если прежнее значение не интересует, во втором параметре можно передать **NULL**.

Чаще всего функция **pthread_setcancelstate()** используется для временного запрета завершения потока. Если в потоке есть участок кода, во время выполнения которого завершать поток крайне нежелательно, то можно оградить этот участок кода от досрочного завершения с помощью пары вызовов **pthread_setcancelstate()**:

```
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
... //Здесь поток завершать нельзя
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
```

Первый вызов **pthread_setcancelstate()** запрещает досрочное завершение потока, второй – разрешает. Если запрос на досрочное завершение потока поступит в тот момент, когда поток игнорирует эти запросы, выполнение запроса будет отложено до тех пор, пока функция **pthread_setcancelstate()** не будет вызвана с аргументом **PTHREAD_CANCEL_ENABLE**.

Интересна роль функции **pthread_testcancel(void)**. Эта функция создает точку отмены потока. Даже если досрочное завершение разрешено, поток, получивший запрос на досрочное завершение, может завершить работу не сразу. Если поток находится в режиме отложенного досрочного завершения (а этот режим установлен по умолчанию), он выполнит запрос на досрочное завершение, только достигнув одной из точек отмены. В соответствии со стандартом POSIX, точками отмены

являются вызовы многих «обычных» функций, например **open()**, **pause()** и **write()**. Про функцию **printf()** в документации сказано, что она может быть точкой отмены, но в Linux при попытке остановиться на **printf()** поток завершается, а **pthread_join()** не возвращает управления. Поэтому создается явная точка отмены с помощью вызова **pthread_testcancel()**.

Можно выполнить досрочное завершение потока, не дожидаясь точек останова. Для этого необходимо перевести поток в режим немедленного завершения с помощью вызова **pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,NULL)**. В этом случае беспокоиться о точках останова уже не нужно. Вызов **pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL)** снова переводит поток в режим отложенного досрочного завершения.

### 7.1.3. Особенности главного потока

Программа на C начинается с выполнения функции **main()**. Поток, в котором выполняется данная функция, называется главным или начальным (так как это первый поток в приложении). Этот поток обладает многими свойствами обычного потока, для него можно получить идентификатор, он может быть отсоединен, для него можно вызвать **pthread_join** из какого-либо другого потока. Но он обладает и некоторыми особенностями, отличающими его от других потоков.

Возврат из этого потока завершает весь процесс. Если не надо, чтобы по завершении этого потока остальные потоки были уничтожены, то следует воспользоваться функцией **pthread_exit**.

У функции этого потока не один параметр типа **void***, как у остальных, а пара **argc-argv**.

Многие реализации отводят на стек начального потока больше памяти, чем на стеки остальных потоков. Это связано с тем, что существует много традиционных, однопоточных приложений, требующих значительного объема стека.

### 7.1.4. Жизненный цикл потоков

Рассмотрим жизненный цикл потока, а именно последовательность состояний, в которых пребывает поток за время своего существования. В целом (табл. 7.1) можно выделить четыре таких состояния.

Любой создаваемый поток начинает свою жизнь в состоянии «готов». После чего в зависимости от политики диспетчеризации задач в операционной системе он может либо сразу перейти в состояние «выполняется», либо перейти в него через некоторое время.

Типичной ошибкой является считать, что (в отсутствие явных мер по синхронизации потоков) после возврата из функции **pthread_create** новый поток будет существовать. Но при некоторых политиках планирования и атрибутах потока может статься, что новый поток успеет выполниться к моменту возврата из этой функции.

Таблица 7.1

**Состояния потока**

| Состояние потока | Что означает |
|---|---|
| Готов (Ready) | Поток готов к выполнению, но ожидает процессора. Возможно, он только что был создан, был вытеснен из процессора другим потоком или только что был разблокирован |
| Выполняется (Running) | Поток сейчас выполняется. Следует заметить, что на многопроцессорной машине может быть несколько потоков в таком состоянии |
| Заблокирован (Blocked) | Поток не может выполняться, так как ожидает чего-либо. Например, окончания операции ввода-вывода |
| Завершен (Terminated) | Поток была завершен, например, вследствие возврата из функции потока, вызова **pthread_exit**. Поток не был отсоединен и для него не была вызвана функция **pthread_join**. Как только происходит одно из этих событий, поток перестает существовать |

### 7.1.5. Атрибуты потоков

Атрибуты являются способом определить поведение потока, отличное от поведения по умолчанию. При создании потока с помощью **pthread_create()** или при инициализации переменной синхронизации может быть определен собственный объект атрибутов. Атрибуты определяются только во время создания потока; они не могут быть изменены в процессе использования.

Обычно вызываются три функции:

- инициализация атрибутов потока - **pthread_attr_init()** создает объект атрибутов **tattr** типа **pthread_attr_t** со значением по умолчанию;
- изменение значений атрибутов (если значения по умолчанию не подходят) – разнообразные функции **pthread_attr_*()**, позволяющие установить значения индивидуальных атрибутов для структуры **pthread_attr_t tattr**;
- создание потока – вызов **pthread_create()** с соответствующими значениями атрибутов в структуре **pthread_attr_t tattr**.

Рассмотрим пример кода (thr_attr.c), выполняющего эти действия.

Объект атрибутов является закрытым и не может быть изменен операциями присваивания. Как только атрибут инициализируется и конфигурируется, он доступен всему процессу. Поэтому рекомендуется конфигурировать все требуемые спецификации состояния один раз на ранних стадиях выполнения программы. При этом соответствующий объект атрибутов может использоваться везде, где это нужно.

Использование объектов атрибутов имеет два преимущества:

1. Это обеспечивает мобильность кода. Даже в случае, когда поддерживаемые атрибуты могут измениться в зависимости от реализации, не нужно будет изменять вызовы функций, которые создают объекты потоков, потому что объект атрибутов скрыт от интерфейса. Задача портирования облегчается, потому что объекты атрибутов будут инициализироваться однажды и в определенном месте.

2. Упрощается спецификация состояний в приложении. Пусть в процессе существует несколько множеств потоков, при этом каждое обеспечивает отдельный сервис и имеет свои собственные требования к состоянию. В некоторый момент на ранних стадиях приложения можно инициализировать объект атрибутов потока для каждого множества. Все будущие вызовы создания потока будут обращаться к объекту атрибутов, инициализированному для этого типа потока.

Функция **pthread_attr_init()** используется, чтобы инициализировать объект атрибутов значениями по умолчанию. Память распределяется системой потоков во время выполнения.

Пример вызова функции:

```
#include <pthread.h>
pthread_attr_t tattr;
```

```
int ret;
ret = pthread_attr_init(&tattr);
```

Функция возвращает 0 после успешного завершения. Любое другое значение указывает, что произошла ошибка. Код ошибки устанавливается в переменной **errno**. Значения по умолчанию для атрибутов приведены в таблице.

Таблица 7.2

**Атрибуты потоков**

| Атрибут | Значение по умолчанию | Назначение |
|---|---|---|
| **detachstate** | **PTHREAD_CREATE_JOINABLE** | Управление состоянием потока (присоединяемый **PTHREAD_CREATE_JOINABLE** / отсоединяемый **PTHREAD_CREATE_DEACHED**) |
| **schedpolicy** | **SCHED_OTHER** | Выбор политики диспетчеризации: **SCHED_OTHER** (non-realtime), **SCHED_RR** (realtime) или **SCHED_FIFO** (realtime) |
| **schedparam** | 0 | Приоритет при диспетчеризации, имеет смысл только для **SCHED_RR** и **SCHED_FIFO** |
| **inheritsched** | **PTHREAD_EXPLICIT_SCHED** | Параметры диспетчеризации потока задаются (**PTHREAD_EXPLICIT_ SCHED)** или наследуются от родительского (**PTHREAD_ INHERIT_SCHED**) |
| **scope** | **PTHREAD_SCOPE_PROCESS** | **PTHREAD_SCOPE_SYSTEM** – поток конкурирует за процессор со всеми потоками системы.<br>**PTHREAD_SCOPE_PROCESS** – поток конкурирует за процессор с потоками, созданными родительским потоком |

Функция **pthread_attr_destroy()** используется, чтобы удалить память для атрибутов, выделенную во время инициализации. Объект атрибутов становится недействительным.

```
ret = pthread_attr_init(&tattr);
. . .
ret = pthread_attr_destroy(&tattr);
```

Функция возвращает 0 после успешного завершения или любое другое значение в случае ошибки.

Если поток создается отделенным (**PTHREAD_CREATE_DETACHED**), его идентификатор и другие ресурсы освобождаются, как только он завершится. Для создания такого потока необходимо перед его созданием вызвать функцию **pthread_attr_setdetachstate()**. По умолчанию поток создается неотделенным (**PTHREAD_CREATE_JOINABLE**) и предполагается, что создающий поток будет ожидать его завершения и выполнять **pthread_join()**. Независимо от типа потока, процесс не закончится, пока не завершатся все потоки. **pthread_attr_setdetachstate()** возвращает 0 после успешного завершения или другое значение в случае ошибки.

```
pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
```

Функция **pthread_attr_getdetachstate()** позволяет определить состояние при создании потока, т. е. был ли он отделенным или присоединяемым. Она возвращает 0 после успешного завершения или любое другое значение в случае ошибки.

```
pthread_attr_getdetachstate (&tattr, &detachstate);
```

Поток может быть не ограничен адресным пространством процесса (имеет тип **PTHREAD_SCOPE_SYSTEM**) или ограничен (имеет тип **PTHREAD_SCOPE_PROCESS**). Оба этих типа доступны только в пределах данного процесса. Функция **pthread_attr_setscope()** позволяет создать потоки указанных типов. **pthread_attr_setscope()** возвращает 0 после успешного завершения или любое другое значение в случае ошибки.

Функция **pthread_attr_getscope()** используется для определения ограниченности потока. **pthread_att_getscope()** воз-

вращает 0 после успешного завершения или другое значение в случае ошибки.

```
ret = pthread_attr_getscope (&tattr, &scope);
```

Стандарт POSIX определяет следующие значения атрибута планирования: **SCHED_FIFO** (First In – First Out, первым пришел – первым вышел), **SCHED_RR** (Round Robin, карусельная), или **SCHED_OTHER** (метод приложения). Дисциплины **SCHED_FIFO** и **SCHED_RR** поддерживаются только для потоков в режиме реального времени. Попытка установить их в других режимах приведет к возникновению ошибки **ENOSUP**.

```
ret=pthread_attr_setschedpolicy
(&tattr, SCHED_OTHER);
```

Парная функция **pthread_attr_getschedpolicy()** возвращает константу, определяющую текущую дисциплину диспетчеризации.

Функция **pthread_attr_setinheritsched()** используется для наследования дисциплины диспетчеризации из родительского потока. Если атрибут **inherit = PTHREAD_ INHERIT_SCHED** (по умолчанию), то будет использована дисциплина планирования родителя. Если используется константа **PTHREAD_EXPLICIT_SCHED**, используются атрибуты, переданные в вызове **pthread_create()**. Функция возвращает 0 при успешном завершении и любое другое значение в случае ошибки:

```
pthread_attr_setinheritsched(&tattr,PTHREAD_EXPLICI
T_SCHED);
```

Функцию **pthread_attr_getinheritsched (&tattr, int *inherit)** можно использовать для получения информации о текущей дисциплине планирования потока.

Параметры диспетчеризации определены в структуре **sched_param**; сейчас поддерживается только приоритет **sched_param.sched_priority**. Этот приоритет задается целым числом, и чем выше его значение, тем выше приоритет потока при планировании. Создаваемые потоки получают этот приоритет. Функция **pthread_attr_setschedparam()** используется, чтобы уста-

новить значения в этой структуре. При успешном завершении она возвращает 0.

```
sched_param param;
param.sched_priority = 20;
ret = pthread_attr_setschedparam (&tattr, &param);
```

Функция **pthread_attr_getschedparam (pthread_attr_t *tattr, const struct sched_param *param)** используется для получения приоритета текущего потока.

## 7.2. Средства синхронизации потоков в Linux

*Взаимные исключения* (mutual exclusion – mutex) и *условные переменные* (conditional variables) являются основными средствами синхронизации действий нескольких программных потоков или процессов. Обычно это требуется для предоставления нескольким потокам или процессам совместного доступа к данным.

Взаимные исключения и условные переменные появились в стандарте POSIX.l для программных потоков и всегда могут быть использованы для синхронизации отдельных *потоков* одного процесса. Стандарт POSIX также разрешает использовать взаимное исключение или условную переменную также для синхронизации нескольких *процессов*, если это исключение или переменная хранится в области памяти, совместно используемой процессами.

Применение взаимных исключений и условных переменных иллюстрируется классической задачей «производитель – потребитель». В примере используются программные потоки, а не процессы, поскольку предоставить потокам общий буфер данных, предполагаемый в этой задаче, легко, а вот создать буфер данных между процессами можно только с помощью одной из форм разделяемой памяти. Другое решение этой задачи возможно с использованием семафоров.

Взаимное исключение (mutex) является простейшей формой синхронизации. Оно используется для защиты *критической области* (critical region), предотвращая одновременное выполнение участка кода несколькими потоками или процессами:

- блокировать_mutex(…);
- критическая область;
- разблокировать_mutex(…).

Поскольку только один поток может заблокировать взаимное исключение в любой момент времени, это гарантирует, что только один поток будет выполнять код, относящийся к критической области.

Переменные-мьютексы определяются с типом **pthread_mutex_t**. Прежде чем использовать переменную-мьютекс, следует сначала инициализировать ее, записав значение константы **PTHREAD_MUTEX_INITIALIZER** (только для статически размещаемых мьютексов) или вызвав функцию **pthread_mutex_init**.

При динамическом выделении памяти под взаимное исключение (например, вызовом **malloc**) или при помещении его в разделяемую память необходимо инициализировать эту переменную во время выполнения, вызвав функцию **pthread_mutex_init()**. Прежде чем освободить занимаемую память, необходимо вызвать функцию **pthread_mutex_destroy**:

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t mutex,
const pthread_mutexattr_t attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Обе функции возвращают 0 в случае успеха, код ошибки – в случае неудачи.

Чтобы инициализировать мьютекс со значениями атрибутов по умолчанию, нужно передать **NULL** в аргументе **attr**. Конкретные значения атрибутов мьютексов мы рассмотрим позже.

Следующие три функции используются для установки и снятия блокировки взаимного исключения:

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mptr);
int pthread_mutex_trylock (pthread_mutex_t *mptr);
int pthread_mutex_unlock (pthread_mutex_t *mptr);
```

Все три возвращают 0 в случае успешного завершения или положительное значение **Exxx** в случае ошибки.

При попытке заблокировать взаимное исключение, которое уже заблокировано другим потоком, функция **pthread_mutex_lock** будет ожидать его разблокирования, а **pthread_mutex_trylock** (неблокируемая функция) вернет ошибку с кодом **BUSY**.

Один из дополнительных примитивов управления мьютексами позволяет ограничить время блокировки потока при попытке захватить

мьютекс, запертый другим потоком. Функция **pthread_mutex_timedlock** эквивалентна функции **pthread_mutex_lock**, но по истечении указанного тайм-аута **pthread_mutex_timedlock** вернет код ошибки **ETIMEDOUT**, не захватывая мьютекс.

```
#include <pthread.h>
#include <time.h>
int pthread_mutex_timedlock(pthread_mutex_t
*restrict mutex,
const struct timespec *restrict tsptr);
```

Функция возвращает 0 в случае успеха, код ошибки **ETIMEDOUT** – в случае неудачи.

Взаимное исключение обычно используется для защиты данных, совместно используемых несколькими потоками или процессами, и представляет собой блокировку *коллективного пользования*. Это значит, что все потоки, работающие с данными, должны блокировать взаимное исключение. Ничто не может помешать потоку работать с данными, не заблокировав взаимное исключение. Взаимные исключения предполагают добровольное сотрудничество потоков.

Одна из классических задач на синхронизацию называется задачей производителей и потребителей (писателей и читателей). Она также известна как задача ограниченного буфера. Один или несколько производителей (потоков или процессов) создают данные, которые обрабатываются одним или несколькими потребителями. Эти данные передаются между производителями и потребителями с помощью одной из форм IPC. Схема задачи изображена на рис. 7.1.

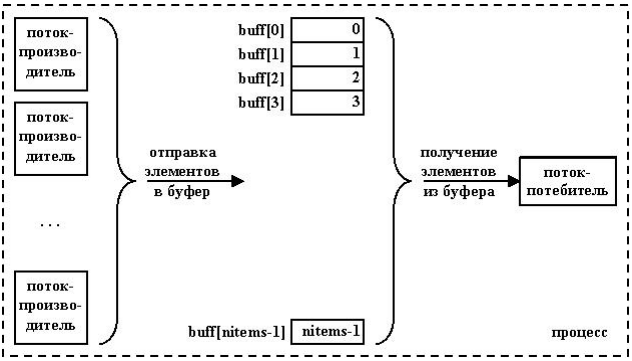



*Рис. 7.1*. Схема задачи производителей – потребителей

В одном процессе у нас имеется несколько потоков-производителей и один поток-потребитель. Целочисленный массив **buff** содержит производимые и потребляемые данные (данные совместного пользования). Для простоты производители просто устанавливают значение **buff[0]** в 0, **buff[1]** в 1 и т. д. Потребитель перебирает элементы массива, проверяя правильность записей.

В первом примере важна синхронизация между потоками-производителями. Поток-потребитель не будет запущен, пока все производители не завершат свою работу. В программе prodcons_on.c, доступной на сайте [6] в папке «Mutex», приведен текст примера.

Буфер **buff** и переменные **nput**, **nval** совместно используются потоками. Они объединены в структуру **shared** вместе с взаимным исключением, чтобы подчеркнуть, что доступ к буферу и переменным можно получить только вместе с взаимным исключением. Переменная **nput** хранит индекс следующего элемента массива **buff**, подлежащего обработке, а **nval** содержит следующее значение, которое должно быть в него помещено. Под структуру выделяется память и инициализируется взаимное исключение, используемое для синхронизации потоков-производителей.

Первый аргумент командной строки указывает количество элементов, которые будут произведены производителями, а второй – количество запускаемых потоков-производителей.

Каждый из создаваемых потоков-производителей вызывает функцию **produce**. Идентификаторы потоков хранятся в массиве **tid_produce**. Аргументом каждого потока-производителя является указатель на элемент массива **count**. Счетчики инициализируются значением 0, и каждый поток увеличивает значение своего счетчика на 1 при помещении очередного элемента в буфер. Содержимое массива счетчиков затем выводится на экран, так что можно узнать, сколько элементов было помещено в буфер каждым из потоков.

Программа ожидает завершения работы всех потоков-производителей, выводя содержимое счетчика для каждого потока, а затем запускает единственный процесс-потребитель. Таким образом (на данный момент) исключается необходимость синхронизации между потребителем и производителями. По завершении работы потока-потребителя завершается работа всего процесса.

Критическая область кода производителя состоит из проверки на достижение конца массива (завершение работы):

```
if (shared.nput >= nitems)
```

и строк, помещающих очередное значение в массив:

```
shared.buff[shared.nput] = shared.nval;
shared.nput++; shared.nval++;
```

Эта область защищается с помощью взаимного исключения, которое разблокируется после завершения работы с элементами структуры **shared**.

Увеличение элемента **count** (через указатель **arg**) не относится к критической области, поскольку у каждого потока счетчик свой (массив **count** в функции **main**). Эта строка не включается в блокируемую взаимным исключением область. Один из принципов хорошего стиля программирования заключается в минимизации объема кода, защищаемого взаимным исключением.

Потребитель проверяет правильность значений всех элементов массива и выводит сообщение в случае обнаружения ошибки. Эта функция запускается в единственном экземпляре и только после того, как все потоки-производители завершат свою работу, так что надобность в синхронизации отсутствует.

Если убрать из этого примера (см. prodcons_off.c) блокировку с помощью взаимного исключения, он перестанет работать так, как предполагается. Потребитель обнаружит ряд элементов **buff[i]**, значения которых будут отличны от i. Другой вариант его работы: несколько производителей будут формировать одни и те же элементы данных.

Также можно убедиться, что удаление блокировки ничего не изменит, если будет выполняться только один поток.

Продемонстрируем, что взаимные исключения предназначены для блокирования, но не для ожидания. Изменим (см. prodcons1.c) пример так, чтобы потребитель запускался сразу после запуска всех производителей, что позволит ему обрабатывать данные сразу по мере их формирования. Теперь придется синхронизовать потребителя с производителями, чтобы первый обрабатывал только данные, уже сформированные последними.

Начало кода (до объявления функции **main**) не претерпело никаких изменений. Поток-потребитель создается сразу же после создания потоков-производителей. Функция **produce** не изменяется. Функция **consume** вызывает новую функцию **consume_wait**. Единственное изменение в функции **consume** заключается в добавлении вызова **consume_wait** перед обработкой следующего элемента массива.

Функция **consume_wait** должна ждать, пока производители не создадут i-й элемент. Для проверки этого условия производится блокировка взаимного исключения и значение i сравнивается с индексом производителя **nput**. Блокировка необходима, поскольку **nput** может быть изменен одним из производителей в момент его проверки.

Возникает вопрос, что делать, если нужный элемент еще не готов. Здесь мы повторяем операции в цикле, устанавливая и снимая блокировку и проверяя значение индекса. Это называется опросом (spinning или polling) и является лишней тратой времени процессора.

Можно было бы приостановить выполнение процесса на некоторое время, но не известно, на какое. Что действительно нужно – это использовать какое-то другое средство синхронизации, позволяющее потоку или процессу приостанавливать работу, пока не произойдет какое-либо событие.

Взаимное исключение используется для блокирования, а условная переменная – для ожидания. Это два различных средства синхронизации, и оба они нужны. Сами условные переменные защищаются мьютексами. Прежде чем изменить значение такой переменной, поток должен захватить мьютекс. Другие потоки не будут замечать изменений в переменной, пока не попытаются захватить этот мьютекс.

Условная переменная, представленная типом **pthread_cond_t**, должна инициализироваться перед использованием. При статическом размещении переменной ей можно присвоить значение константы **PTHREAD_COND_INITIALIZER**, но если переменная состояния размещается динамически, то ее следует инициализировать вызовом **pthread_cond_init**. Для уничтожения условной переменной перед освобождением занимаемой ею памяти используется функция **pthread_cond_destroy**:

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t cond,
const pthread_condattr_t attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Обе функции возвращают 0 в случае успеха, код ошибки – в случае неудачи.

Для работы с условными переменными предназначены две функции:

```
#include <pthread.h>
```

```
int pthread_cond_wait(pthread_cond_t *cptr,
pthread_mutex_t *mptr);
int pthread_cond_signal(pthread_cond_t *cptr);
```

Обе функции возвращают 0 в случае успешного завершения, положительное значение **Exxx** – в случае ошибки. Слово «signal» в имени второй функции не имеет никакого отношения к сигналам UNIX **SIGxxx**. Просто определяется условие, уведомления о выполнении которого поток будет ожидать.

В программе prodcons2.c две переменные **nput** и **nval** ассоциируются с **mutex**, и мы объединяем их в структуру с именем **put**. Эта структура используется производителями.

Другая структура, **nready**, содержит счетчик, условную переменную и взаимное исключение, инициализируемое с помощью **PTHREAD_COND_INITIALIZER**. Функция **main** по сравнению с предыдущим листингом не изменяется.

Функции **produce** и **consume** претерпевают некоторые изменения. Для блокирования критической области в потоке-производителе теперь используется исключение **put.mutex**. Там же увеличивается счетчик **nready.nready**, в котором хранится количество элементов, готовых для обработки потребителем. Перед его увеличением проверяется, было ли значение счетчика нулевым, и если нет, то вызывается функция **pthread_cond_signal**, позволяющая возобновить выполнение всех потоков (здесь – потребителя), ожидающих установки ненулевого значения этой переменной.

Счетчик используется совместно потребителем и производителями, поэтому доступ к нему осуществляется с блокировкой соответствующего взаимного исключения (**nready.mutex**).

Потребитель просто ждет, пока значение счетчика **nready.nready** не станет отличным от нуля. Поскольку этот счетчик используется совместно с производителями, его значение можно проверять только при блокировке соответствующего взаимного исключения. Если при проверке значение оказывается нулевым, вызывается **pthread_cond_wait** для приостановки процесса.

При этом выполняются два атомарных действия.

1. Разблокируется **nready.mutex**.

2. Выполнение потока приостанавливается, пока какой-нибудь другой поток не вызовет **pthread_cond_signal**.

Перед возвращением управления потоку функция **pthread_cond_wait** блокирует **nready.mutex**. Если после возвращения из функции обнаруживается, что счетчик имеет ненулевое значение, то этот счетчик уменьшается (зная, что взаимное исключение заблокировано) и разблокируется взаимное исключение. После возвращения из **pthread_cond_wait** всегда заново проверяется условие, поскольку может произойти ложное пробуждение.

В примере кода функция **pthread_cond_signal** вызывалась потоком, блокировавшим взаимное исключение, относящееся к условной переменной, для которой отправлялся сигнал. В худшем варианте система немедленно передаст управление потоку, которому направляется сигнал, и он начнет выполняться и немедленно остановится, поскольку не сможет заблокировать взаимное исключение.

Исправленный код, помогающий этого избежать, будет иметь вид:

```
int dosignal;
pthread_mutex_lock(nready.mutex);
doslgnal= (nready.nready == 0);
nready.nready++;
pthread_mutex_unlock(&nready.mutex);
if (doslgnal)
pthread_cond_signal(&nready.cond);
```

Здесь сигнал условной переменной отправляется только после разблокирования взаимного исключения. Это разрешено стандартом POSIX: поток, вызывающий **pthread_cond_signal**, не обязательно должен в этот момент блокировать связанное с переменной взаимное исключение. Однако POSIX говорит, что если требуется предсказуемое поведение при одновременном выполнении потоков, это взаимное исключение должно быть заблокировано процессом, вызывающим **pthread_cond_signal**.

В обычной ситуации **pthread_cond_signal** запускает выполнение одного потока, ожидающего сигнал по соответствующей условной переменной. В некоторых случаях поток знает, что требуется пробудить несколько других процессов. Тогда можно воспользоваться функцией **pthread_cond_broadcast** для пробуждения всех процессов, заблокированных в ожидании сигнала данной условной переменной.

```
#include <pttread.h>
int pthread_cond_broadcast (pthread_cond_t *aptr);
```

```
int pthread_cond_timewait (pthread_cond_t, *cptr,
pthread_mutex_t *mpfr, const struct timespec
*abstime);
```

Функции возвращают 0 в случае успешного завершения или положительный код **Exxx** в случае ошибки.

Функция **pthread_cond_timedwait** позволяет установить ограничение на время блокирования процесса. Аргумент **abstime** представляет собой структуру **timespec**:

```
struct timespec {
ttme_t tv_sec; /* секунды */
long tv_nsec; /*наносекунды. Заявлен на перспективу */
};
```

Эта структура задает конкретный момент системного времени, в который происходит возврат из функции, даже если сигнал по условной переменной еще не будет получен. В этом случае возвращается ошибка с кодом **ETIMEDOUT**. Этот момент представляет собой абсолютное значение времени, а не промежуток. Аргумент **abstime** задает количество секунд (наносекунды реально не поддерживаются) с 00:00 UTC 1 января 1970 г. до того момента времени, в который должен произойти возврат из функции.

Это отличает функцию от **select**, **pselect** и **poll**, принимающих в качестве аргумента некоторое количество долей секунды, спустя которое должен произойти возврат. (Функция **select** принимает количество секунд и микросекунд, **pselect** – секунд, а **poll** – миллисекунд.) Преимущество использования абсолютного времени заключается в том, что если функция возвратится до ожидаемого момента (например, при перехвате сигнала), то ее можно будет вызвать еще раз, не изменяя содержимого структуры **timespec**.

Атрибуты взаимного исключения имеют тип **pthread_mutexattr_t**, а условной переменной – **pthread_condattr_t**, и инициализируются и уничтожаются с помощью следующих функций:

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t
*attr);
int pthread_mutexattr_destroy (pthread_mutexattr_t
*attr);
```

```
int pthread_condattr_init (pthread_condattr_t
*attr);
int pthread_condattr_destroy (pthread_condattr_t
*attr);
```

Все четыре функции возвращают 0 в случае успешного завершения или положительное значение **Exxx** в случае ошибки.

После инициализации объекта атрибутов для включения или выключения отдельных атрибутов используются отдельные функции. Один из атрибутов позволяет использовать взаимное исключение или условную переменную нескольким процессам. Его значение можно узнать и изменить с помощью следующих функций:

```
#include <pthread.h>
int pthread_mutexattr_getpshared(const
pthread_mutexattr_t *attr, int *valptr);
int pthread_mutexattr_setpshared
(pthread_muutexattr_t *attr, int value);
int pthread_condattr_getpshared(const
pthread_condattr_t *attr, int *valptr);
int pthread_condattr_setpshared (pthread_condattr_t
*attr, int value);
```

Все четыре функции возвращают 0 в случае успешного завершения или положительное значение **Exxx** в случае ошибки.

Две функции **get*** возвращают текущее значение атрибута через целое, на которое указывает **valptr**, а две функции **set*** устанавливают значение атрибута равным значению **value**. Значение **value** может быть либо **PTHREAD_PROCESS_PRIVATE**, либо **PTHREAD_PROCESS_SHARED**. Последнее также называется атрибутом совместного использования процессами.

Вот как нужно инициализировать взаимное исключение для совместного использования несколькими процессами:

```
pthread_mutex_t *mptr;
pthread_mutexattr_t mattr;
mptr = malloc(sizeof (pthread_mutex_t));/* адрес */
pthread_mutexattr_init(&mattr);
pthread mutexattr_setpshared (&mattr,
PTHREAD_PROCESS_SHARED);
pthread_mutex_init(mptr, &mattr);
```

Здесь объявляется переменная **mattr** типа **pthread_mutexattr_t**, инициализируется значениями атрибутов по умолчанию, а затем устанавливается атрибут **PTRREAD_PROCESS_SHARED**, позволяющий нескольким процессам совместно использовать взаимное исключение. Затем **pthread_mutex_init** инициализирует само исключение с соответствующими атрибутами.

Такая же последовательность команд (с заменой **mutex** на **cond**) позволяет установить атрибут **PTHREAD_PROCESS_SHARED** для условной переменной, хранящейся в разделяемой процессами памяти.

Когда взаимное исключение используется совместно несколькими процессами, всегда существует возможность, что процесс будет завершен (возможно, принудительно) во время работы с заблокированным им ресурсом. В Windows API такие мьютексы называют *покинутыми*. В Linux не существует способа заставить систему автоматически снимать блокировку с ресурса во время завершения процесса. Единственный тип блокировок, автоматически снимаемых системой при завершении процесса, – это блокировки записей **fcntl**.

Поток также может быть завершен в момент работы с заблокированным ресурсом, если его выполнение отменит (**pthread_cancel**) другой поток или он сам вызовет **pthread_exit**. Последнее сомнительно, поскольку поток должен сам знать, блокирует ли он взаимное исключение в данный момент или нет, и в зависимости от этого вызывать **pthread_exit**. На случай отмены другим потоком можно предусмотреть обработчик сигнала, вызываемый при отмене потока. Если же для потока возникают фатальные условия, это обычно приводит к завершению работы всего процесса.

Даже если бы система автоматически разблокировала ресурсы после завершения процесса, это не всегда решало бы проблему. Блокировка защищала критическую область, в которой, возможно, изменялись какие-то данные. Если процесс был завершен посреди этой области, что стало с данными? Велика вероятность того, что возникнут несоответствия. Если бы ядро просто разблокировало взаимное исключение при завершении процесса, следующий обратившийся к списку процесс обнаружил бы, что тот поврежден.

## Вопросы для самопроверки

1. Опишите функцию порождения потока и особенности функции, запускаемой в порожденном потоке.
2. Каковы особенности POSIX API для работы с потоками?
3. Когда и как происходит завершение потока?
4. Как сделать поток отсоединенным?
5. Опишите особенности досрочного завершения потока.
6. Каковы особенности главного потока в процессе?
7. Опишите жизненный цикл потоков.
8. Зачем нужен и как создается объект атрибутов потока?
9. Каковы значения атрибутов потока по умолчанию?
10. Какие функции применяются для задания/чтения атрибутов потока?
11. Что такое мьютекс и зачем он нужен?
12. Когда и как необходимо инициализировать мьютекс?
13. Опишите функции для работы с мьютексами и их особенности.
14. Что такое условные переменные и зачем они нужны?
16. Какие функции применяются для работы с условными переменными?
17. В каких случаях при использовании условных переменных возникает ложное пробуждение?
18. Для чего применяются функции **`pthread_cond_broadcast`** и **`pthread_cond_timewait`**?
19. Каковы особенности инициализации мьютексов и условных переменных, расположенных в разделяемой памяти?
20. Какие существуют значения атрибутов мьютекса? Что они дают?

## Упражнения

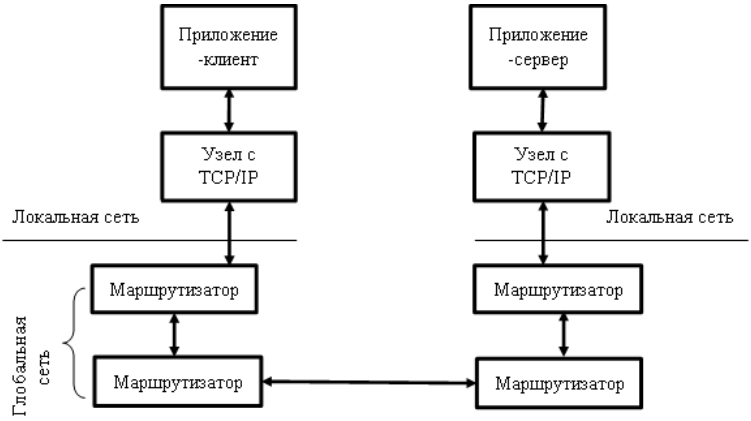
Выполните лабораторную работу № 7 из лабораторного практикума.

# Глава 8. Сетевое взаимодействие процессов в Linux

Чтобы писать программы, рассчитанные на взаимодействие в компьютерных сетях, необходимо сначала изучить сетевые протоколы – соглашения о порядке взаимодействия таких программ. Кроме того, нужно принять решение о том, какая программа будет инициировать передачу данных и в каких случаях можно ожидать ответной передачи. Например, веб-сервер обычно рассматривается как долгоживущая программа (или *демон* – *daemon*), которая отправляет сообщения исключительно в ответ на запросы, поступающие по сети. Другой стороной является веб-клиент, например браузер, который всегда начинает взаимодействие с сервером первым. Деление на клиенты и серверы характерно для большинства сетевых приложений.

Клиенты обычно устанавливают соединение с одним сервером за один раз, хотя, если в качестве примера говорить о веб-браузере, мы можем соединиться со множеством различных веб-серверов. Сервер, в свою очередь, в любой момент времени может быть соединен со множеством клиентов.

Хотя клиентское и серверное приложения взаимодействуют по сетевому протоколу, фактически в большинстве случаев используется несколько протоколов различных уровней. Далее сосредоточимся на наборе (стеке) протоколов TCP/IP, также называемом набором протоколов Интернета. Так, например, клиенты и веб-серверы устанавливают соединения, используя протокол управления передачей (Transmission Control Protocol, TCP). TCP в свою очередь использует протокол Интернета (Internet Protocol, IP), а протокол IP устанавливает соединение с тем или иным протоколом канального уровня. Если и клиент, и сервер находятся в одной сети Ethernet, взаимодействие между ними будет осуществляться по схеме [9], изображенной на рисунке.




Клиент и сервер в одной сети Ethernet, соединенные по протоколу TCP

Хотя клиент и сервер устанавливают соединение с использованием протокола уровня приложений, транспортные уровни устанавливают соединение, используя TCP. Обратите внимание, что действительный поток информации между клиентом и сервером идет вниз по стеку протоколов на стороне клиента, затем по сети и, наконец, вверх по стеку протоколов на стороне сервера. Заметим, что клиент и сервер являются типичными пользовательскими процессами, в то время как TCP и протоколы IP обычно являются частью стека протоколов внутри ядра (рис. 8.1).

Некоторые клиенты и серверы используют протокол пользовательских дейтаграмм (User Datagram Protocol, UDP) вместо TCP. Используя термин «IP», подразумеваем протокол «IP версии 4» (IP version 4, IPv4). Новая версия этого протокола, IP версии 6 (IPv6), была разработана в середине 90-х и, возможно, со временем заменит протокол IPv4. Далее описана разработка сетевых приложений под IPv4.

## 8.1. Сокеты, дейтаграммы и потоки передачи

В локальных и глобальных сетях существует два принципиально разных способа передачи данных.

Первый из них предполагает посылку пакетов данных от одного узла другому (или сразу нескольким узлам) без получения подтвер-

ждения о доставке и даже без гарантии того, что передаваемые пакеты будут получены в правильной последовательности. Примером такого протокола может служить протокол UDP. Основные преимущества протокола заключаются в высоком быстродействии и возможности широковещательной передачи данных, когда один узел отправляет сообщения, а другие их получают, причем все одновременно.

Второй способ передачи данных предполагает установление соединения для передачи данных между двумя различными узлами сети. При этом соединение создается средствами дейтаграммных протоколов, однако доставка пакетов в канале является гарантированной. Пакеты всегда доходят в целостности и сохранности, причем в правильном порядке, хотя быстродействие получается в среднем ниже за счет посылки подтверждений. Примером протокола, использующего соединение, служит протокол TCP.

Для передачи данных с использованием любого из перечисленных выше способов каждое приложение должно создать объект, который называется сокетом. По своему назначению сокет больше всего похож на дескриптор файла, который нужен для выполнения над файлом операций чтения или записи. Прежде чем приложение, запущенное на узле сети, сможет выполнять передачу или прием данных, оно должно создать сокет и проинициализировать его, указав некоторые параметры.

```
#include <sys/socket.h>
int socket(int family , int type , int protocol );
```

Функция возвращает неотрицательный дескриптор, если функция выполнена успешно, и –1 в случае ошибки.

Параметр **family** обозначает семейство адресов, или протокол; для указания протокола IPv4 следует указать **AF_INET**, для IPv6 – **AF_INET6**, а для протокола доменных сокетов UNIX – **AF_LOCAL** (или **AF_UNIX**, если не используется стандарт POSIX).

Параметр **type** указывает тип взаимодействия:

- **SOCK_STREAM** – ориентированное на установку соединения по протоколу TCP (или потоковое);
- **SOCK_DGRAM** – рассчитанное на использование протокола UDP (дейтаграммное);
- **SOCK_RAW**) – использование сырых, неструктурированных сокетов (рассмотрение которых выходит за пределы данного пособия).

Параметр **protocol** является излишним, поскольку он может принимать значения **IPPROTO_TCP** или **IPPROTO_UDP**, а тип взаимодействия уже установлен параметром **type**; если параметр **family** установлен равным **AF_INET**, то можно использовать значение 0.

Пример создания сокета:

```
int sockfd;
sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

## 8.2. Серверные функции сокета

В нижеследующем обсуждении под *сервером* будет пониматься процесс, который принимает запросы на образование соединения через заданный порт на заданном сетевом устройстве (интерфейсе).

### 8.2.1. Связывание сокета

Связывание заключается в привязке сокета к адресу сетевого интерфейса и *номеру порта* (port number) на нем. Любой заданный сервер может иметь несколько открытых портов на одном интерфейсе. Функция **bind** связывает сокет с локальным адресом протокола. В случае протоколов Интернета адрес протокола – это комбинация 32-разрядного адреса IPv4 или 128-разрядного адреса IPv6 с 16-разрядным номером порта TCP или UDP. Прототип функции **bind:**

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *saddr,
int namelen);
```

Здесь **sockfd** – несвязанный сокет, возвращенный функцией **socket**, структура **saddr** заполняется перед вызовом и задает протокол и специфическую для протокола информацию, как описано ниже. Кроме всего прочего, в этой структуре содержится номер порта. Параметр **namelen** должен быть равен значению **sizeof (sockaddr)**.

В случае успеха функция возвращает значение 0, в случае ошибки –1.

С помощью функции **bind** процесс может связать конкретный IP-адрес с сокетом. IP-адрес должен соответствовать одному из интерфейсов узла. При этом для сервера TCP на сокет накладывается огра-

ничение: он может принимать только те входящие соединения клиента, которые предназначены именно для этого IP-адреса.

Клиент TCP может не связывать IP-адрес с сокетом при помощи функции **bind**. Ядро выбирает IP-адрес отправителя в момент подключения клиента к сокету, основываясь на используемом исходящем интерфейсе, который, в свою очередь, зависит от маршрута, требуемого для обращения к серверу.

Структура **sockaddr** определяется следующим образом:

```
<netinet/in.h>
struct sockaddr {
u_short sa_family;
char sa_data[14] ;
};
```

Первый член этой структуры, **sa_family**, обозначает протокол. Второй член, **sa_data**, зависит от протокола. Internet-версией структуры **sa_data** является структура **sockaddr_in**:

```
struct sockaddr_in {
short sin_family; /* AF_INET */
unsigned short sin_port;
struct in_addr sin_addr; /* 4-байтовый IP-адрес */
char sin_zero[8];
} sa;
```

О связанном сокете, для которого определены протокол, номер порта и IP-адрес, иногда говорят как об *именованном сокете* (named socket).

Пример связывания сокета:

```
sockaddr_in local_addr;
local_addr.sin_family=AF_INET;  // задание системы
адресации
local_addr.sin_port=htons(PORT); // задание порта
local_addr.sin_addr.s_addr=INADDR_ANY;// подключения
со всех IP-адресов
bind(socket,(sockaddr *) &local_addr,
sizeof(local_addr));
```

Номер порта и IP-адрес должны храниться с соблюдением сетевого порядка следования байтов. При таком порядке старший байт помеща-

ется в крайней позиции справа (big-endian), чтобы обеспечивалась двоичная совместимость с другими системами. Для смены порядка следования байтов из обычного (порядок хоста, little-endian) в сетевой формат (big-endian) применяются функции **htons()** (Host to Network short) и **htonl()** (Host to Network long). Обратное преобразование выполняется функциями **ntohs()** и **ntohl()** соответственно.

В структуре **in_addr** содержится подструктура **sin_addr**, заполняемая 4-байтовым IP-адресом интерфейса, например любого: **INADDR_ANY**. Для работы на локальном компьютере можно использовать значение **INADDR_LOOPBACK** (127.0.0.1);

Дейтаграммный протокол UDP позволяет посылать пакеты данных одновременно всем рабочим станциям в широковещательном режиме. Для этого вы должны указать адрес как **INADDR_BROADCAST**.

Для преобразования текстовой строки с конкретным IP-адресом к требуемому формату можно использовать функцию **inet_addr**, поэтому член **in_addr.sin_addr** переменной **sockaddr_in** инициализируется следующим образом:

```
sa.in_addr.sin_addr = inet_addr("192.168.0.1");
```

В случае ошибки функция возвращает значение **EADDRINUSE**, указывающее на то, что адрес уже используется. Обратное преобразование адреса IP в текстовую строку можно при необходимости легко выполнить с помощью функции **inet_ntoa**, имеющей следующий прототип:

```
char * inet_ntoa (struct in_addr in);
```

При ошибке эта функция возвращает значение **NULL**.

### 8.2.2. Функции работы с DNS

Чаще всего пользователь сетевого приложения указывает не IP-адреса, а доменные имена серверов, при этом неявно используется сервер DNS (сервер доменных имен, Domain Name Server). Чтобы обеспечить пользователю такую возможность, программисту необходимо воспользоваться функцией **gethostbyname:**

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
```

Она получает доменное имя хоста в виде строки и возвращает структуру с его описанием типа **hostent**:

```
struct hostent {
    char    *h_name;
    char    **h_aliases;
    int     h_addrtype;
    int     h_length;
    char    **h_addr_list;
};
#define h_addr h_addr_list[0]
```

Здесь:

- **h_name** – имя хоста.
- **h_aliases** – массив строк, содержащих псевдонимы хоста. Завершается значением **NULL**.
- **h_addrtype** – тип адреса. Для Internet-домена – **AF_INET**.
- **h_length** – длина адреса в байтах.
- **h_addr_list** – массив, содержащий адреса всех сетевых интерфейсов хоста. Завершается нулем. Обратите внимание, что байты каждого адреса хранятся в сетевом порядке, поэтому функцию **htonl** вызывать не нужно.

Затем следует записать полученный адрес в структуру **in_addr:**

```
hostent hst;
hst = gethostbyname ("ftp.microsoft.com");
if (hst)
memcpy((char *)&(dest_sin.sin_addr ), hst ->h_addr,
hst ->h_length);
```

В случае ошибки функция **gethostbyname** возвращает **NULL**. При этом расширенный код ошибки записывается в глобальную переменную **h_errno** (а не **errno**). Соответственно, для вывода диагностического сообщения следует использовать **herror** вместо **perror**.

Если же указанный узел найден в базе DNS, функция **gethostbyname** возвращает указатель на структуру **hostent**. Искомый адрес находится в этой структуре в первом элементе списка **h_addr_list[0]**, на который можно также ссылаться при помощи **h_addr**. Длина поля адреса находится в поле **h_length**.

Для обратного действия – определения имени хоста по адресу – используется функция **gethostbyaddr**. Вместо строки она получает адрес (в виде **sockaddr**) и возвращает указатель на ту же самую структуру **hostent**:

```
#include <netdb.h>
struct hostent *gethostbyaddr(char *addr, int len,
int type);
```

Здесь же рассмотрим еще одно семейство полезных функций – **gethostname**, **getsockname** и **getpeername**:

```
#include <unistd.h>
int gethostname(char *hostname, size_t size);
#include <sys/socket.h>
int getsockname(int s, struct sockaddr *name,
socklen_t *namelen);
#include <sys/socket.h>
int getpeername(int sockfd, struct sockaddr *addr,
int *addrlen);
```

Функция **gethostname** используется для получения имени локального хоста. Далее его можно преобразовать в адрес при помощи **gethostbyname**.

Функция **getsockname** возвращает текущее имя указанного сокета в параметре **name**. В параметре **namelen** должно быть указано, сколько места выделено под **name**. При возврате в этом параметре передается реальный использованный размер в байтах.

Функция **getpeername** позволяет в любой момент узнать адрес сокета на «другом конце» соединения. Она получает дескриптор сокета, соединенного с удаленным хостом, и записывает адрес этого хоста в структуру, на которую указывает **addr**. Фактическое количество записанных байтов помещается по адресу **addrlen** (не забудьте записать туда размер структуры **addr** до вызова **getpeername**). Полученный адрес при необходимости можно преобразовать в строку, используя **inet_ntoa** или **gethostbyaddr**. Функция **getsockname** по назначению обратна **getpeername** и позволяет определить адрес сокета на «нашем конце» соединения.

### 8.2.2. Перевод связанного сокета в состояние прослушивания

Функция **listen** вызывается только сервером TCP и выполняет два действия.

1. Функция **listen** преобразует неприсоединенный сокет в пассивный сокет, запросы на подключение к которому начинают приниматься ядром.

2. Второй аргумент этой функции задает максимальное число соединений, которые ядро может помещать в очередь этого сокета.

```
#include <sys/socket.h>
int listen(int sockfd , int backlog );
```

Возвращает 0 в случае успешного выполнения, –1 в случае ошибки.

Эта функция обычно вызывается после функций **socket** и **bind**. Она должна вызываться перед вызовом функции **accept**.

Аргумент **backlog** функции **listen** исторически задавал максимальное значение для очереди соединений. Традиционно в примерах кода всегда используется значение **backlog**, равное 5, поскольку это было максимальное значение, которое поддерживалось в системе 4.2BSD. В настоящее время многие системы позволяют администраторам изменять максимальное значение аргумента **backlog**. Всегда можно задавать значение больше того, которое поддерживается ядром, так как ядро должно обрезать значение до максимального, не возвращая при этом ошибку.

### 8.2.3. Прием клиентских запросов соединения

Сервер может ожидать соединения с клиентом, используя функцию **accept**, возвращающую новый подключенный сокет, который будет использоваться в операциях ввода-вывода. Заметьте, что исходный (привязанный) сокет, который теперь находится в состоянии прослушивания (listening state), используется исключительно в качестве параметра функции **accept**, а не для непосредственного участия в операциях ввода-вывода.

Функция **accept** вызывается сервером TCP для возвращения следующего установленного соединения из начала очереди соединений. Если очередь полностью установленных соединений пуста, процесс переходит в состояние ожидания (по умолчанию предполагается блокируемый сокет).

```
#include <sys/socket.h>
int accept(int sockfd , struct sockaddr *cliaddr ,
socklen_t *addrlen );
```

Функция возвращает неотрицательный дескриптор в случае успешного выполнения или –1 в случае ошибки.

Возможно создание неблокирующихся сокетов, но рассмотрение этого выходит за рамки данного пособия.

Аргументы **cliaddr** и **addrlen** используются для возвращения адреса протокола подключившегося процесса (клиента). Перед вызовом мы присваиваем целому числу, на которое указывает ***addrlen**, размер структуры адреса сокета, на которую указывает аргумент **cliaddr:**

```
addrlen = sizeof(struct sockaddr_in);
```

По завершении функции это целое число содержит действительное число байтов, помещенных ядром в структуру адреса сокета.

Если выполнение функции **accept** прошло успешно, она возвращает новый дескриптор, автоматически созданный ядром. Этот дескриптор используется для обращения к соединению TCP с конкретным клиентом. Значение, возвращаемое этой функцией, называют *присоединенным сокетом* (*connected socket*). Сервер обычно создает только один прослушиваемый сокет, который существует в течение всего времени жизни сервера. Затем ядро создает по одному присоединенному сокету для каждого клиентского соединения, принятого с помощью функции **accept**. Когда сервер заканчивает предоставление сервиса данному клиенту, сокет закрывается.

Если вам не нужно, чтобы был возвращен адрес клиента, следует сделать указатели **cliaddr** и **addrlen** пустыми указателями.

### 8.2.4. Отключение и закрытие сокетов

Обычная функция UNIX **close** также используется для закрытия сокета и завершения соединения TCP:

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

По умолчанию функция **close** помечает сокет TCP как закрытый и немедленно возвращает управление процессу. Дескриптор сокета больше не используется процессом и не может быть передан в каче-

стве аргумента функции **read** или **write**. Но TCP попытается отправить данные, которые уже установлены в очередь, и после их отправки осуществит нормальную последовательность завершения соединения TCP.

## 8.3. Клиентские функции сокета

Клиентская станция, которая желает установить соединение с сервером, также должна создать сокет, вызвав функцию **socket**. Следующий шаг заключается в установке соединения с сервером, а кроме того, необходимо указать номер порта, адрес хоста и другую информацию. Имеется только одна дополнительная функция – **connect**:

```
#include <sys/socket.h>
int connect(int sockfd , const struct sockaddr
*servaddr , socklen_t addrlen );
```

Функция возвращает 0 в случае успешного выполнения, –1 в случае ошибки.

Аргумент **sockfd** – это дескриптор сокета, возвращенный функцией **socket**. Второй и третий аргументы – это указатель на структуру адреса сокета и ее размер. Структура адреса сокета должна содержать IP-адрес и номер порта сервера. Клиенту нет необходимости вызывать функцию **bind** до вызова функции **connect**: при необходимости ядро само выберет и динамически назначаемый порт, и IP-адрес отправителя.

## 8.4. Отправка и получение данных

Программы, использующие сокеты, обмениваются данными по протоколу TCP с помощью функций **send** и **recv**. Эти две функции аналогичны стандартным функциям **read** и **write**, но для них требуется дополнительный аргумент:

```
#include <sys/socket.h>
ssize_t recv(int sockfd, void *buff, size_t nbytes,
int flags );
ssize_t send(int sockfd, const void *buff, size_t
nbytes, int flags );
```

Обе функции возвращают количество прочитанных или записанных байтов в случае успешного выполнения, –1 в случае ошибки.

Первые три аргумента функций **recv** и **send** совпадают с тремя первыми аргументами функций **read** и **write**. Аргумент **flags** либо имеет нулевое значение, либо формируется в результате применения операции логического ИЛИ к константам, представленным в таблице.

**Аргумент flags для функций ввода-вывода**

| flags | Описание | recv | send |
|---|---|---|---|
| **MSG_DONTROUTE** | Не искать в таблице маршрутизации | | • |
| **MSG_OOB** | Отправка или получение срочных данных | • | |
| **MSG_PEEK** | Просмотр приходящих сообщений | • | |
| **MSG_WAITALL** | Ожидание всех данных | • | |

**MSG_DONTROUTE**. Этот флаг сообщает ядру, что получатель находится в нашей сети и поэтому не нужно выполнять поиск в таблице маршрутизации.

**MSG_OOB**. С функцией **send** этот флаг указывает, что отправляются внеполосные (Out Of Band) данные, которые иногда называются *срочными данными* (*expedited data*). Суть этой концепции заключается в том, что если на одном конце соединения происходит какое-либо важное событие, то требуется быстро сообщить об этом собеседнику. В данном случае «быстро» означает, что сообщение должно быть послано прежде, чем будут посланы какие-либо обычные данные (называемые иногда *данными из полосы пропускания*), которые уже помещены в очередь для отправки, то есть внеполосные данные имеют более высокий приоритет, чем обычные данные. Для передачи внеполосных данных не создается новое соединение, а используется уже существующее. В случае TCP в качестве внеполосных данных должен быть отправлен только один байт. С функцией **recv** этот флаг указывает на то, что вместо обычных данных должны читаться внеполосные данные.

**MSG_PEEK**. Этот флаг позволяет просмотреть пришедшие данные, готовые для чтения, при этом после выполнения функции **recv** или **recvfrom** данные не сбрасываются (при повторном вызове этих функций снова возвращаются уже просмотренные данные).

**MSG_WAITALL**. Он сообщает ядру, что операция чтения должна выполняться до тех пор, пока не будет прочитано запрашиваемое количество байтов. Даже если мы задаем флаг **MSG_WAITALL**, функция может возвратить количество байтов меньше запрашиваемого в том случае, если или перехватывается сигнал, или соединение завершается, или есть ошибка сокета, требующая обработки.

Дейтаграммный сокет (типа **SOCK_DGRAM**) также может пользоваться функциями **send** и **recv**, если предварительно вызовет **connect**, но у него есть и свои, «персональные», функции:

```
int sendto (SOCKET s, const char * buf, int len,int
flags, struct sockaddr * to, int tolen);
int recvfrom (SOCKET s, char * buf, int len, int
flags, struct sockaddr * from, int * fromlen);
```

Они очень похожи на **send** и **recv**, – разница лишь в том, что **sendto** и **recvfrom** требуют явного указания адреса узла, принимающего или передающего данные. Вызов **recvfrom** не требует предварительного задания адреса передающего узла – функция принимает все пакеты, приходящие на заданный UDP-порт со всех IP адресов и портов. Напротив, отвечать отправителю следует на тот же самый порт, откуда пришло сообщение. Поскольку функция **recvfrom** запоминает IP-адрес и номер порта клиента после получения от него сообщения, программисту нужно передать в **sendto** тот же самый указатель на структуру **sockaddr**, который был ранее передан функции **recvfrom**, получившей сообщение от клиента.

Еще одна деталь: транспортный протокол UDP, на который опираются дейтаграммные сокеты, не гарантирует успешной доставки сообщений, и эта задача ложится на плечи самого разработчика. Решить ее можно, например, посылкой клиентом подтверждения об успешности получения данных.

Во всем остальном обе пары функций полностью идентичны и работают с теми же флагами. Все четыре функции при возникновении ошибки возвращают значение **SOCKET_ERROR**.

Все функции возвращают в качестве значения функции длину данных, которые были прочитаны или записаны. При типичном использовании функции **recvfrom** с протоколом дейтаграмм возвращаемое значение – это объем пользовательских данных в полученной дейтаграмме.

Дейтаграмма может иметь нулевую длину. В случае UDP при этом возвращается дейтаграмма IP, содержащая заголовок IP (обычно 20 байт для IPv4 или 40 байт для IPv6), 8-байтовый заголовок UDP и никаких данных. Это также означает, что возвращаемое из функции **`recvfrom`** нулевое значение вполне приемлемо для протокола дейтаграмм: оно не является признаком того, что собеседник закрыл соединение, как это происходит при возвращении нулевого значения из функции **`recv`** на сокете TCP. Поскольку протокол UDP не ориентирован на установление соединения, то в нем и не существует такого события, как закрытие соединения.

Примеры применения сокетов, реализующие клиент-серверную систему с использованием протоколов TCP и UDP, доступны на сайте дисциплины [6] в папке «Linsockets».

## Вопросы для самопроверки

1. Опишите функцию создания сокета и ее параметры.
2. Зачем необходимо и как выполняется связывание сокета?
3. Опишите порядки записи IP-адреса и номера порта, а также функции для их преобразования.
4. Какие поименованные константы можно использовать при связывании сокета?
5. Опишите функции для преобразования доменного имени в IP-адрес и обратно.
6. Перечислите дополнительные функции преобразования имен и адресов.
7. Как происходит перевод связанного сокета сервера в состояние прослушивания?
8. Как производится прием клиентских запросов сервером?
9. Как происходит подключение клиента к серверу?
10. Опишите функции отправки и получения данных по протоколу TCP.
11. Опишите функции отправки и получения данных по протоколу UDP.
12. Перечислите флаги функций отправки и получения данных.

## Упражнения

Выполните лабораторную работу № 8 из лабораторного практикума.

# Глава 9. Системные службы (демоны) в Linux

*Демон* (англ. *daemon*) – это процесс, обладающий следующими свойствами.

- Имеет длинный жизненный цикл. Часто демоны создаются во время загрузки системы и работают до момента ее выключения.
- Выполняется в фоновом режиме и не имеет контролирующего терминала.

Последняя особенность гарантирует, что ядро не сможет генерировать для такого процесса никаких сигналов, связанных с терминалом или управлением заданиями (таких, как **`SIGINT`**, **`SIGHUP`**).

Демоны создаются для выполнения специфических задач. Например:

- **`cron`** – демон, который выполняет команды в запланированное время;
- **`sshd`** – демон защищенной командной оболочки, который позволяет входить в систему с удаленных компьютеров, используя безопасный протокол;
- **`httpd`** – демон HTTP-сервера (Apache), который обслуживает веб-страницы;

Многие стандартные демоны работают в качестве привилегированных процессов (то есть их действующий пользовательский идентификатор равен 0), поэтому при их написании следует руководствоваться рекомендациями по написанию безопасных программ с повышенными привилегиями.

## 9.1. Создание демона

Для того чтобы стать демоном, программа должна выполнить следующие шаги.

1. Сделать вызов **`fork()`**, после которого родитель завершается, а потомок продолжает работать (в результате этого демон становится

потомком процесса **init**). Этот шаг делается по двум следующим причинам.

- Исходя из того, что демон был запущен в командной строке, завершение родителя будет обнаружено командной оболочкой, которая вслед за этим выведет новое приглашение и позволит потомку выполняться в фоновом режиме.
- Потомок гарантированно не станет лидером группы процессов, поскольку он наследует идентификатор группы программ **PGID** от своего родителя и получает свой уникальный идентификатор, который отличается от унаследованного **PGID**. Это необходимо для успешного выполнения следующего шага.

2. Дочерний процесс вызывает **setsid()**, чтобы начать новую сессию и разорвать любые связи с контролирующим терминалом.

*Контролирующий терминал* – это тот, который устанавливается при первом открытии устройства терминала лидером сессии. Любой контролирующий терминал может быть связан не более чем с одной сессией. *Сессия* – это набор групп процессов. Членство процесса в сессии определяется идентификатором SID (session identifier – *идентификатор сессии*), который по аналогии с PGID (process group identifier – *идентификатор группы процессов*) является числом типа **pid_t.** *Лидером сессии* является процесс, который ее создал и чей идентификатор используется в качестве SID. Новый процесс наследует идентификатор SID своего родителя. Этот вызов возвращает идентификатор новой сессии или –1, если случилась ошибка:

```
#include <unistd.h>
pid_t setsid(void);
```

Создание новой сессии системным вызовом **setsid()** происходит следующим образом:

- вызывающий процесс становится лидером новой сессии и новой группы процессов внутри нее. Идентификаторы **PGID** и **SID** нового процесса получают то же значение, что и сам процесс;
- вызывающий процесс не имеет контролирующего терминала. Любое соединение с контролирующим терминалом, установленное ранее, разрывается.

3. Если после этого демон больше не открывает никаких терминальных устройств, мы можем не волноваться о том, что он восстановит соединение с контролирующим терминалом. В противном случае

нам необходимо сделать так, чтобы терминальное устройство не стало контролирующим. Это можно сделать двумя нижеописанными способами.

- указывать флаг **O_NOCTTY** для любых вызовов **open()**, которые могут открыть терминальное устройство.
- более простой вариант: после **setsid()** можно еще раз сделать вызов **fork()**, опять позволив родителю завершиться, а потомку (правнуку) – продолжить работу. Это гарантирует, что потомок не станет лидером сессии, что делает невозможным повторное соединение с контролирующим терминалом (это соответствует процедуре получения контролирующего терминала, принятой в System V).

4. Очистить атрибут **umask** процесса, чтобы файлы и каталоги, созданные демоном, имели запрашиваемые права доступа.

5. Поменять текущий рабочий каталог процесса (обычно на корневой – /). Это необходимо, поскольку демон обычно выполняется вплоть до выключения системы. Если файловая система, на которой находится его текущий рабочий каталог, не является корневой, она не может быть отключена. Как вариант, в качестве рабочего каталога демон может задействовать то место, где он выполняет свою работу, или воспользоваться значением в конфигурационном файле; главное, чтобы файловая система, в которой находится этот каталог, никогда не нуждалась в отключении. Например, **cron** применяет для этого **/var/spool/cron**.

6. Закрыть все открытые файловые дескрипторы, которые демон унаследовал от своего родителя (возможно, некоторые из них необходимо оставить открытыми, поэтому данный шаг является необязательным и может быть откорректирован). Это делается по целому ряду причин. Поскольку демон потерял свой контролирующий терминал и работает в фоновом режиме, ему больше не нужно хранить дескрипторы с номерами 0, 1 и 2 (они ссылаются на терминал). Кроме того, мы не можем отключить файловую систему, на которой долгоживущий демон удерживает открытыми какие-либо файлы. И, следуя обычным правилам, мы должны закрывать неиспользуемые файловые дескрипторы, поскольку их число ограниченно.

Рассмотрим пример кода, реализующего вышеописанные действия:

```
switch (fork ()) { /* Превращение в фоновый процесс */
case -1: return -1;
case 0: break; / * Потомок проходит этот этап . . . * /
```

```
default: _exit(EXIT_SUCCESS); /* . . . а родитель
завершается */
}
if ( setsid () == -1 ) / * Процесс становится лидером
новой сессии * /
return -1;
switch (fork ()) { / * Делаем так, чтобы процесс
не стал лидером сессии * /
case -1: return -1;
case 0: break;
default : _exit(EXIT_SUCCESS);
}
umask(0); / * Сбрасываем маску режима создания файлов *
/
chdir ( " / " ) ; / * Переходим в корневой каталог * /
/ * Закрываем все открытые файлы * /
maxfd = sysconf (_SC_OPEN_MAX);
if (maxfd == -1 ) / * Ограничение не определено... * /
maxfd = BD_MAX_CLOSE; / * ...поэтому устанавливаем
его наугад */
for ( fd = 0; fd < maxfd; fd ++ )
close (fd) ;
close(STDIN_FILENO); / *Перенаправляем стандартные
потоки данных в /dev/null */
fd = open ("/dev/null", 0_RDWR);
if ( fd != STDIN_FILENO) /* Значение fd должно быть
больше 0 */
return -1;
if (dup2(STDIN_FILENO, STDOUT_FILENO) !=
STDOUT_FILENO)
return - 1;
if (dup2(STDIN_FILENO, STDERR_FILENO) !=
STDERR_FILENO)
return -1;
```

## 9.2. Функция daemon( )

Библиотека GNU C предоставляет нестандартную функцию **daemon()**, которая превращает вызывающий процесс в демона.

Функция **daemon()** необходима для того, чтобы отключить программу от управляющего терминала и запустить ее в фоновом режиме подобно тому, как выполняются системные службы.

```
#include <unistd.h>
int daemon(int nochdir, int noclose);
```

Если аргумент **nochdir** равен нулю, то **daemon()** изменяет текущий рабочий каталог процесса на корневой («/»); в противном случае текущий рабочий каталог не изменяется.

Если аргумент **noclose** равен нулю, то **daemon()** перенаправляет стандартный поток ввода, вывода и ошибок в **/dev/null**; в противном случае данные файловые дескрипторы не изменяются.

Эта функция порождает новый процесс и, если **fork()** завершается без ошибок, родительский процесс вызывает **_exit()**, чтобы дальнейшие ошибки воспринимались только дочерним процессом. В случае успешного выполнения **daemon()** возвращает ноль. Если возникла ошибка, то **daemon()** возвращает –1 и присваивает глобальной переменной **errno** одно из значений, указанных для **fork()** и **setsid()**.

Для библиотеки GNU C реализация этой функции была взята из BSD, и в ней не применяется техника двойного **fork** (т. е. **fork()**, **setsid()**, **fork()**), поэтому необходимо проверить, что полученный процесс службы не является лидером сеанса. Вместо этого полученная служба *является* лидером сеанса. В системах, следующих семантике System V (например, Linux), это означает, что если служба открывает терминал, который пока не является управляющим для другого сеанса, то этот терминал непреднамеренно станет управляющим терминалом для службы.

## 9.3. Запись в журнал сообщений и ошибок с помощью системы syslog

При написании демона одной из проблем является вывод сообщений об ошибках. Поскольку демон выполняется в фоновом режиме, он не может выводить информацию в терминале, как это делают другие

программы. В качестве альтернативы сообщения можно записывать в отдельный журнальный файл программы.

Для записи сообщений в журнал любой процесс может воспользоваться библиотечной функцией **syslog()**. На основе переданных ей аргументов она создает сообщение стандартного вида и помещает его в сокет **/dev/log**, где оно будет доступно для **syslogd**.

Программный интерфейс **syslog** состоит из трех основных функций.

1. Функция **openlog()** устанавливает настройки, которые по умолчанию применяются ко всем последующим вызовам **syslog()**. Она не является обязательной. Если ею не воспользоваться, соединение с системой ведения журнала устанавливается при первом вызове **syslog()** на основе стандартных настроек.

2. Функция **syslog()** записывает сообщения в журнал.

3. Функция **closelog()** вызывается после окончания записи сообщений, чтобы разорвать соединение с журналом.

Ни одна из этих функций не возвращает значение статуса. Частично это продиктовано тем, что системное журналирование должно быть всегда доступным (если оно перестанет работать, системный администратор должен быстро это заметить). Кроме того, если при ведении журнала произошла ошибка, приложение обычно мало что может сделать, чтобы об этом сообщить.

Функция **closelog()** закрывает описатель, используемый для записи данных в журнал. Использование **closelog()** необязательно.

Функция **openlog()** при необходимости устанавливает соединение с системным средством ведения журнала и задает настройки, которые будут применяться по умолчанию ко всем последующим вызовам **syslog()**.

```
#include <syslog.h>
void openlog (const char *ident, int log_options,
int facility);
```

Аргумент **ident** является указателем на строку, которая добавляется в каждое сообщение, записываемое с помощью **syslog()**; обычно это название программы. Стоит отметить, что **openlog()** всего лишь копирует значение этого указателя. Продолжая использовать вызовы **syslog()**, приложение должно следить за тем, чтобы строка, на которую ссылается данный аргумент, не изменилась.

Если в качестве аргумента **ident** указать **NULL**, интерфейс **syslog** из состава **glibc**, как и некоторые другие реализации, будет автоматически подставлять вместо него название программы. Однако такое поведение не предусмотрено стандартом SUSv3 и не выполняется в некоторых системах, поэтому переносимые приложения не должны на него полагаться.

Аргумент **log_options** для вызова **openlog()** представляет собой битовую маску, состоящую из любых комбинаций следующих констант, к которым применяется побитовое ИЛИ.

- **LOG_CONS** – если в системный журнал приходит ошибка, она записывается в системную консоль (**/dev/console**).
- **LOG_NDELAY** – соединение с системой ведения журнала (то есть с сокетом домена UNIX, **/dev/log**) устанавливается немедленно. По умолчанию (**LOG_ODELAY**) это происходит, только когда (и если) первое сообщение попадает в журнал с помощью вызова **syslog()**. Флаг **LOG_NDELAY** может пригодиться в программах, которым нужно контролировать момент выделения файлового дескриптора для **/dev/log**. Например, это может быть приложение, которое вызывает **chroot()**; после этого вызова путь **/dev/log** перестает быть доступным, поэтому, если вы вызываете функцию **openlog()** с флагом **LOG_NDELAY**, это нужно делать до **chroot()**. Примером программы, которая использует флаг **LOG_NDELAY** таким образом, может служить демон **tftpd** (*Trivial File Transfer*).
- **LOG_NOWAIT** – вызов **syslog()** не ждет дочерний процесс, который мог быть создан для записи сообщения в журнал. Этот флаг нужен в приложениях, в которых для записи сообщений используются отдельные дочерние процессы. Он позволяет вызову **syslog()** избежать ожидания потомков, уже утилизированных родителем, который тоже их ожидал. В Linux флаг **LOG_NOWAIT** ни на что не влияет, так как в этой системе при записи сообщений в журнал дочерние процессы не создаются.
- **LOG_ODELAY** – противоположность флагу **LOG_NDELAY**. Соединение с системой ведения журнала откладывается до тех пор, пока не будет записано первое сообщение. Этот флаг используется по умолчанию, и его не нужно указывать отдельно.
- **LOG_PID** – включать PID в каждое сообщение.

Аргумент **facility** устаналивает значение по умолчанию, если не указываются соответствующие параметры при вызовах **syslog()**. Аргумент **facility** используется для указания типа программы, записывающей сообщения. Это позволяет файлу конфигурации указывать, что сообщения от различных программ будут по-разному обрабатываться.

- **LOG_AUTH** – сообщения о безопасности/авторизации (рекомендуется использовать вместо него **LOG_AUTHPRIV**);
- **LOG_AUTHPRIV** – сообщения о безопасности/авторизации (частные);
- **LOG_CRON** – демон часов (**cron** и **at**);
- **LOG_DAEMON** – другие системные демоны;
- **LOG_KERN** – сообщения ядра;
- **LOG_LOCAL0** до **LOG_LOCAL7** – зарезервированы для определения пользователем;
- **LOG_LPR** – подсистема принтера;
- **LOG_MAIL** – почтовая подсистема;
- **LOG_NEWS** – подсистема новостей USENET;
- **LOG_SYSLOG** – сообщения, генерируемые **syslogd**;
- **LOG_USER** (по умолчанию) – общие сообщения на уровне пользователя;
- **LOG_UUCP** – подсистема UUCP.

Функция **syslog()** изначально разрабатывалась для BSD, в настоящее время она предоставляется большинством производителей систем UNIX.

```
void syslog(int priority, const char *format, ...);
```

**syslog()** создает сообщение для журнала, которое передается демону **syslogd**. **priority** получается при логическом сложении **facility**, описанном выше, и **level**, описанном ниже. Аргументы **format** такие же, как и в **printf()**, кроме того, что сочетание **%m** будет заменено сообщением об ошибке **strerror(errno)** и будет добавлен завершающий символ новой строки.

Параметр **level** определяет степень важности сообщения. Далее значения приводятся по понижению степени их важности:

- **LOG_EMERG** – система остановлена;
- **LOG_ALERT** – требуется немедленное вмешательство;

- **LOG_CRIT** – критические условия;
- **LOG_ERR** – ошибки;
- **LOG_WARNING** – предупреждения;
- **LOG_NOTICE** – важные рабочие условия;
- **LOG_INFO** – информационные сообщения;
- **LOG_DEBUG** – сообщения об отладке.

Функция **setlogmask()** может использоваться для ограничения доступа на указанные уровни.

Назначение аргументов **facility** и **level** в том, чтобы все сообщения, которые посылаются процессами определенного типа (то есть с одним значением аргумента **facility**), могли обрабатываться одинаково в файле **/etc/syslog.conf** или чтобы все сообщения одного уровня (с одинаковым значением аргумента **level**) обрабатывались одинаково. Например, демон может сделать следующий вызов, когда вызов функции **rename** неожиданно оказывается неудачным:

```
syslog(LOG_INFO|LOG_LOCAL2, "rename(%s, %s): %m",
file1, file2);
```

Конфигурационный файл **/etc/syslog.conf** определяет поведение демона **syslogd** . Он состоит из правил и комментариев (последние начинаются с символа #). Правила в общем случае имеют следующий вид:

```
категория.приоритет          действие
```

Сочетание *категории* и *приоритета* называют *селектором,* поскольку они позволяют выбрать сообщения, к которым применяется правило. Виды категорий (**facility**) и приоритетов (**level**) описаны выше, но в файле конфигурации применяются без префикса **LOG_**. Под *действием* подразумевается место назначения сообщений, которые соответствуют *селектору. Селектор* и *действие* разделены пробельными символами. Ниже показан пример нескольких правил:

```
*.err              /dev/tty10
auth.notice        root
*.debug;mail.none;news.none      -/var/log/messages
```

Согласно первому правилу сообщения всех категорий (*****) с приоритетом **err** (**LOG_ERR**) или выше должны передаваться консольному устройству **/dev/tty10**. Второе правило делает так, что сообщения,

связанные с авторизацией (**LOG_AUTH**) и имеющие приоритет **notice** (**LOG_NOTICE**) или выше, должны отправляться во все консоли или терминалы, в которых работает пользователь **root**. Это, например, позволит администратору немедленно получать все сообщения о неудачных попытках повышения привилегий (вызове команды **su**).

В последней строчке демонстрируются некоторые продвинутые аспекты синтаксиса для описания правил. В ней перечислено сразу несколько селекторов, разделенных точкой с запятой. Первый селектор относится к сообщениям любой категории (*****) с приоритетом **debug** (самым низким) и выше, т. е. это затрагивает все сообщения. В Linux, как и в большинстве других UNIX-систем, вместо **debug** можно указать символ *****, который будет иметь то же значение, однако данная возможность поддерживается не всеми реализациями **syslog**. Если правило содержит несколько селекторов, оно обычно охватывает сообщения, соответствующие любому из них. Но если в качестве приоритета указать значение **none**, то сообщения, принадлежащие к заданной категории, будут отбрасываться. Таким образом, это правило передает все сообщения (кроме тех, которые имеют категории **mail** и **news**) в файл **/var/log/messages**. Символ «тильда» (**~**) перед именем этого файла говорит о том, что сбрасывание данных на диск будет происходить не при каждой передаче сообщения. Это приводит к увеличению скорости записи, но в случае сбоя системы сообщения, пришедшие недавно, могут быть утеряны.

При каждом изменении файла syslog.conf демону следует отправлять сигнал, чтобы он смог заново себя инициализировать:

```
$killall -HUP syslogd //Отправляем сигнал SIGHUP
демону syslogd
```

Синтаксис файла syslog.conf позволяет создавать куда более сложные правила, чем те, что были показаны. Все подробности можно найти на справочной странице syslog.conf [6].

## Вопросы для самопроверки

1. Дайте определение системной службы (демона) UNIX.
2. Перечислите шаги, необходимые для преобразования приложения в демона.
3. Для чего предназначена и что возвращает функция **setsid()**?

4. Дайте определение неконтролирующего терминала, сессии, лидера сессии.

5. Чем функция **daemon()** отличается от набора действий по преобразованию приложения в демона?

6. Каковы входные параметры функции **daemon()**?

7. Перечислите состав программного интерфейса **syslog**.

8. Каковы входные параметры функции **openlog()**?

9. Каковы входные параметры функции **syslog()**?

10. Какова структура конфигурационного файла **/etc/syslog.conf**?

11. Что необходимо делать демону **syslogd** при изменении файла его конфигурации? Как именно?

## Упражнения

Выполните курсовую (расчетно-графическую) работу № 9 из лабораторного практикума.

# Глава 10. Лабораторный практикум

## Лабораторная работа № 1

## Файловые операции средствами системных вызовов

**1. Цель работы**: получить навыки разработки приложений, реализующих операции с файлами средствами системных вызовов Linux API на языке C в операционных системах семейства Linux.

**2. Краткие теоретические сведения**: см. главу 3, стр. 25. Пример кода: листинг 3.1 (см. стр. 27) и файл file.c в разделе «Примеры программ», «Файлы» сайта по дисциплине [6].

**3. Методические указания.**

3.1. Проект может быть реализован в виде консольного приложения в среде ОС Ubuntu, Fedora, CentOS или других средствами компилятора gcc версии не ниже 4.

3.2. Проект должен быть реализован с применением системных вызовов для открытия, чтения, записи и закрытия файла. Ввод исходных данных производится исключительно через аргументы командной строки. Вывод сообщений об ошибках и результатах работы программы может производиться средствами стандартной библиотеки языка C.

3.3. Проект должен предусматривать обработку исключительных ситуаций (отсутствие или неверное количество входных параметров, ошибки открытия входного и/или выходного файла, ошибки чтения и записи).

3.4. Проект может содержать системные вызовы для блокировки файлов (входного – на чтение, выходного – на запись). ***Наличие функций блокировки файлов соответствует заданию повышенной сложности.***

**4. Порядок выполнения работы.**

4.1. Написать и отладить программу, получающую в аргументах командной строки имя существующего текстового файла, имя выходного файла, который будет переписан при его наличии и в который будет помещен результат работы программы и символ, слово или число, используемый (ое) для обработки файла.

4.2. Результатом работы программы является выходной текстовый файл, содержащий текст, обработанный согласно вариантам, и возвращаемое значение – количество выполненных операций или –1 в случае ошибки.

4.3. При обработке исключительных ситуаций сообщения об ошибках выводятся на терминал.

4.4. Результатом выполнения лабораторной работы считается демонстрация работы программы и обработки исключительных ситуаций преподавателю.

**5. Варианты заданий.**

**Варианты заданий**

| Номер варианта | Задание | Параметры командной строки |
|---|---|---|
| 1 | Удалить из текста заданный символ | 1. Имя входного файла<br>2. Заданный символ |
| 2 | В конце каждой строки вставить заданный символ | 1. Имя входного файла<br>2. Заданный символ |
| 3 | Заменить цифры на пробелы | 1. Имя входного файла<br>2. Количество замен |
| 4 | Заменить знаки на заданный символ | 1. Имя входного файла<br>2. Заданный символ |
| 5 | Заменить каждый пробел на два | 1. Имя входного файла<br>2. Количество замен |
| 6 | После каждой точки вставить символ ‘\n’ | 1. Имя входного файла<br>2. Количество замен |
| 7 | Удалить из текста все пробелы | 1. Имя входного файла<br>2. Количество замен |
| 8 | Заменить заданные символы на пробелы | 1. Имя входного файла<br>2. Заданный символ |
| 9 | После каждого пробела вставить точку | 1. Имя входного файла<br>2. Количество вставок |

Окончание таблицы

| Номер варианта | Задание | Параметры командной строки |
|---|---|---|
| 10 | Заменить все пробелы первым символом текста | 1. Имя входного файла<br>2. Максимальное количество замен |
| 11 | Во всех парах одинаковых символов второй символ заменить на пробел | 1. Имя входного файла<br>2. Количество замен |
| 12 | Заменить на пробелы все символы, совпадающие с первым символом в строке | 1. Имя входного файла<br>2. Количество замен |
| 13 | Заменить заданную пару букв на символы #@ | 1. Имя входного файла<br>2. Заданная пара букв |
| 14 | Заменить все цифры заданным символом | 1. Имя входного файла<br>2. Заданный символ |
| 15 | Заменить на пробел все символы, совпадающие с последним символом в строке | 1. Имя входного файла<br>2. Количество замен |
| 16 | Заменить все символы с кодами меньше 48 на пробелы | 1. Имя входного файла<br>2. Количество замен |
| 17 | Заменить все символы с кодами больше 48 на пробелы | 1. Имя входного файла<br>2. Количество замен |
| 18 | Заменить каждый третий символ на пробел | 1. Имя входного файла<br>2. Количество замен |
| 19 | Заменить все пробелы на заданный символ | 1. Имя входного файла<br>2. Заданный символ |
| 20 | Заменить все пары одинаковых символов на пробелы | 1. Имя входного файла<br>2. Количество замен |

**6. Содержание отчета.**

6.1. Цель работы.

6.2. Вариант задания.

6.3. Листинг программы.

**7. Контрольные вопросы**: см. вопросы для самопроверки к главе 3 (стр. 44).

# Лабораторная работа № 2

# Статические и динамические библиотеки

**1. Цель работы**: получить навыки разработки и использования статических и динамических библиотек в программах на языке С в операционных системах семейства Linux.

**2. Краткие теоретические сведения**: см. главу 4, стр. 45. Пример кода: листинги параграфа 4.1 (см. стр. 45), 4.5.4 (см. стр. 62) и файлы main0.c, f1.c, f2,c, main1.c, lib.c в разделе «Примеры программ», подраздел «Библиотеки» сайта по дисциплине [6].

**3. Методические указания.**

3.1. Проект может быть реализован в виде консольного приложения в среде ОС Ubuntu, Fedora, CentOS или других средствами компилятора gcc версии не ниже 4.

3.2. Проект является модификацией лабораторной работы № 1, основная бизнес-логика которого (обработка содержимого входного файла) реализована в виде функции, помещенной в статическую/динамическую библиотеку.

3.3. Проект должен предусматривать обработку исключительных ситуаций (отсутствие или неверное количество входных параметров, ошибки открытия входного и/или выходного файла, ошибки чтения и записи).

3.4. Проект рекомендуется реализовать в 3 этапа.

3.4.1. Вынесение операций обработки содержимого входного файла в функцию, описанную в этом же файле программы.

3.4.2. Вынесение функции в отдельный файл, формирование статической библиотеки, применение статической библиотеки.

3.4.3. Формирование разделяемой библиотеки и модификация головной программы для динамической загрузки и выгрузки библиотеки.

**4. Порядок выполнения работы.**

4.1. Модифицировать программу из лабораторной работы № 1, вынося операции обработки входного файла в функцию.

4.2. Сформировать статическую библиотеку, подключить к головной программе, продемонстрировать работоспособность программы преподавателю.

4.3. Сформировать разделяемую библиотеку, модифицировать головную программу для динамической загрузки разделяемой библиотеки, продемонстрировать работоспособность программы преподавателю.

4.4. Результатом работы каждой версии программы является выходной текстовый файл, содержащий текст, обработанный согласно вариантам, и возвращаемое значение – количество выполненных операций или –1 в случае ошибки.

4.5. При обработке исключительных ситуаций сообщения об ошибках выводятся на терминал.

4.6. Результатом выполнения лабораторной работы считается демонстрация работы двух вариантов программы (со статической и динамической библиотеками) и обработки исключительных ситуаций преподавателю.

**5. Варианты заданий**: см. лабораторную работу № 1 (стр. 214).

**6. Содержание отчета.**

6.1. Цель работы.

6.2. Вариант задания.

6.3. Листинги программ.

**7. Контрольные вопросы**: см. вопросы для самопроверки к главе 4 (стр. 64).

# Лабораторная работа № 3

# Многозадачное программирование в Linux

**1. Цель работы**: получить навыки разработки многозадачных приложений на языке C в операционных системах семейства Linux.

**2. Краткие теоретические сведения**: см. главу 5, стр. 71. Пример кода: файлы parent.c и child.c в разделе «Примеры программ из пособия», подраздел «Многозадачные приложения» сайта по дисциплине [6].

**3. Методические указания.**

3.1. Проект может быть реализован в виде консольного приложения в среде ОС Ubuntu, Fedora, CentOS или других средствами компилятора gcc версии не ниже 4.

3.2. Проект требует написания родительского приложения для запуска программы из лабораторной работы № 2 для обработки каждого из нескольких входных файлов.

3.3. Проект должен содержать функции порождения дочернего процесса (в нужном количестве экземпляров), запуска в каждом дочернем процессе программы из лабораторной работы № 2 с необходимыми ей аргументами командной строки, ожидания завершения каждого дочернего процесса и получения его кода завершения.

3.4. Проект должен предусматривать обработку исключительных ситуаций (недостаточное количество аргументов командной строки, невозможность запуска дочернего процесса).

**4. Порядок выполнения работы.**

4.1. Написать программу, запускающую программу из лабораторной работы № 1 в количестве экземпляров, соответствующем количеству входных файлов, ждущую завершения всех экземпляров и получающую (выводящую на терминал) коды их завершения.

4.2. Результатом работы программы являются выходные текстовые файлы, содержащие текст, обработанный согласно вариантам, и возвращаемые дочерними процессами значения – количество выполненных операций или –1 в случае ошибки.

4.3. При обработке исключительных ситуаций сообщения об ошибках выводятся на терминал.

4.4. Результатом выполнения лабораторной работы считается демонстрация работы родительской программы с несколькими входными файлами и обработки исключительных ситуаций преподавателю.

**5. Варианты заданий**: см. лабораторную работу № 1 (стр. 214).

**6. Содержание отчета.**

6.1. Цель работы.

6.2. Вариант задания.

6.3. Листинг программы.

**7. Контрольные вопросы**: см. вопросы для самопроверки к главе 5 (стр. 84).

# Лабораторная работа № 4

# Каналы передачи данных

**1. Цель работы**: получить навыки организации обмена информацией между процессами средствами неименованных или именованных каналов в программах на языке С в операционных системах семейства Linux.

**2. Краткие теоретические сведения:** см. параграф 6.2, стр. 87. Примеры кода: файлы pipe_parent.c и pipe_child.c для программных каналов, либо server_main.c и client_main.c для каналов FIFO в разделе «Примеры программ из пособия», подраздел «Pipes» сайта по дисциплине [6].

**3. Методические указания.**

3.1. Проект может быть реализован в виде консольного приложения в среде ОС Ubuntu, Fedora, CentOS или других средствами компилятора gcc версии не ниже 4.

3.2. Проект является модификацией лабораторной работы № 3, в которой обмен данными между родительской и дочерними программами производится через программные каналы.

3.3. Проект может быть реализован в виде многозадачного приложения при использовании программных каналов или клиент-серверного приложения при использовании каналов FIFO. ***Второй способ соответствует заданию повышенной сложности.***

3.4. Проект должен предусматривать обработку исключительных ситуаций (отсутствие или неверное количество входных параметров, ошибки открытия входного и/или выходного файла, ошибки чтения и записи).

3.5. Проект должен предусматривать обмен данными между родительской и дочерними программами или между клиентом и сервером только через каналы (программные или FIFO соответственно).

**4. Порядок выполнения работы:**

4.1. В соответствии с личными предпочтениями выбрать средство реализации межзадачных коммуникаций.

4.2. Модифицировать и отладить в соответствии с выбранным средством коммуникации программу из лабораторной работы № 3, реализующую порожденный (дочерний) процесс. При необходимости модифицировать ее в отдельную программу-сервер.

4.3. Модифицировать и отладить в соответствии с выбранным средством коммуникации программу из лабораторной работы № 3, реализующую родительский процесс. При необходимости модифицировать ее в отдельную программу-клиент.

4.4. Результатом работы программы является совокупность выходных текстовых файлов, соответствующих совокупности (2 и более) входных файлов, содержащих текст, обработанный согласно вариантам, и возвращаемое значение – количество выполненных операций или –1 в случае ошибки.

4.5. При обработке исключительных ситуаций сообщения об ошибках выводятся на терминал.

4.6. Результатом выполнения лабораторной работы считается демонстрация работы программы и обработки исключительных ситуаций преподавателю.

**5. Варианты заданий**: см. лабораторную работу № 1 (стр. 214).

**6. Содержание отчета.**

6.1. Цель работы.

6.2. Вариант задания.

6.3. Листинг программы.

**7. Контрольные вопросы**: см. вопросы для самопроверки к разделу 6.2 (стр. 97).

# Лабораторная работа № 5
# Очереди сообщений

**1. Цель работы**: получить навыки организации обмена данными между процессами средствами очередей сообщений (System V или POSIX) в программах на языке C в операционных системах семейства Linux.

**2. Краткие теоретические сведения**: см. параграф 6.3.2, стр. 105 (System V) или 6.4.2, стр. 131 (POSIX). Примеры кода: файлы в разделе «SysVmsg» или файлы раздела «PosixMsg» сайта по дисциплине [6].

**3. Методические указания.**

3.1. Проект может быть реализован в виде консольного приложения в среде ОС Ubuntu, Fedora, CentOS или других средствами компилятора gcc версии не ниже 4.

3.2. Проект является модификацией лабораторной работы № 4, в которой обмен данными между родительской и дочерней программами (либо клиентом и сервером) производится через очереди сообщений.

3.3. Проект может быть реализован с применением очередей сообщений System V или POSIX. ***Второй способ соответствует заданию повышенной сложности.***

3.4. Проект должен предусматривать обработку исключительных ситуаций (отсутствие или неверное количество входных параметров, ошибки создания очереди сообщений, ошибки чтения и записи из/в нее).

3.5. Проект должен предусматривать обмен данными между родительской и дочерними программами или между клиентом и сервером только через очереди сообщений (System V или POSIX).

**4. Порядок выполнения работы.**

4.1. В соответствии с личными предпочтениями выбрать средство реализации очередей сообщений (System V или POSIX).

4.2. Модифицировать и отладить в соответствии с выбранным средством коммуникации программу из лабораторной работы № 4, реализующую порожденный (дочерний) процесс. При необходимости модифицировать ее в отдельную программу-сервер.

4.3. Модифицировать и отладить в соответствии с выбранным средством коммуникации программу из лабораторной работы № 4, реализующую родительский процесс. При необходимости модифицировать ее в отдельную программу-клиент.

4.4. Результатом работы программы является совокупность выходных текстовых файлов, соответствующих совокупности (2 и более) входных файлов, содержащих текст, обработанный согласно вариантам, и возвращаемое значение – количество выполненных операций или –1 в случае ошибки.

4.5. При обработке исключительных ситуаций сообщения об ошибках выводятся на терминал.

4.6. Результатом выполнения лабораторной работы считается демонстрация работы программы (программ) и обработки исключительных ситуаций преподавателю.

**5. Варианты заданий**: см. лабораторную работу № 1 (стр. 214).

**6. Содержание отчета.**

6.1. Цель работы.

6.2. Вариант задания.

6.3. Листинг программы.

**7. Контрольные вопросы**: см. вопросы для самопроверки к разделу 6.3 (стр. 123) или 6.4 (стр. 163) в зависимости от выбранного средства реализации очередей сообщений (System V или POSIX).

# Лабораторная работа № 6

# Семафоры и разделяемая память

**1. Цель работы**: получить навыки организации обмена данными между процессами средствами разделяемой памяти с синхронизацией посредством семафоров (System V или POSIX) в программах на языке С в операционных системах семейства Linux.

**2. Краткие теоретические сведения**: см. параграф 6.3.3, стр. 112 и параграф 6.3.4, стр. 116 (System V) или 6.4.3, стр. 144 и 6.4.4, стр. 153 (POSIX). Примеры кода: файлы разделов «SysV_sem», «SysV_shm» или файлы разделов «PosixSem», «PosixShm» сайта по дисциплине [6].

**3. Методические указания.**

3.1. Проект может быть реализован в виде консольного приложения в среде ОС Ubuntu, Fedora, CentOS или других средствами компилятора gcc версии не ниже 4.

3.2. Проект является модификацией лабораторной работы № 5, в которой обмен данными между родительской и дочерней программами (либо клиентом и сервером) производится через разделяемую память, а синхронизация программ – посредством семафоров.

3.3. Проект может быть реализован с применением очередей сообщений System V или POSIX. ***Второй способ соответствует заданию повышенной сложности***.

3.4. Проект должен предусматривать обработку исключительных ситуаций (отсутствие или неверное количество входных параметров, ошибки создания семафоров и разделяемой памяти, ошибки управления семафорами, чтения и записи из/в разделяемую память).

3.5. Проект должен предусматривать обмен данными между родительской и дочерними программами или между клиентом и сервером только через разделяемую память (System V или POSIX).

**4. Порядок выполнения работы.**

4.1. В соответствии с личными предпочтениями выбрать средство реализации очередей сообщений (System V или POSIX).

4.2. Модифицировать и отладить в соответствии с выбранным средством коммуникации программу из лабораторной работы № 5, реализующую порожденный (дочерний) процесс. При необходимости модифицировать ее в отдельную программу-сервер.

4.3. Модифицировать и отладить в соответствии с выбранным средством коммуникации программу из лабораторной работы № 5, ре-

ализующую родительский процесс. При необходимости модифицировать ее в отдельную программу-клиент.

4.4. Результатом работы программы является совокупность выходных текстовых файлов, соответствующих совокупности (2 и более) входных файлов, содержащих текст, обработанный согласно вариантам, и возвращаемое значение – количество выполненных операций или –1 в случае ошибки.

4.5. При обработке исключительных ситуаций сообщения об ошибках выводятся на терминал.

4.6. Результатом выполнения лабораторной работы считается демонстрация работы программы (программ) и обработки исключительных ситуаций преподавателю.

**5. Варианты заданий**: см. лабораторную работу № 1 (стр. 214).

**6. Содержание отчета.**

6.1. Цель работы.

6.2. Вариант задания.

6.3. Листинг программы.

**7. Контрольные вопросы**: см. вопросы для самопроверки к разделу 6.3 (стр. 124) или 6.4 (стр. 163) в зависимости от выбранного средства реализации очередей сообщений (System V или POSIX).

# Лабораторная работа № 7

# Многопоточное программирование в Linux

**1. Цель работы**: получить навыки разработки многопоточных приложений на языке C в операционных системах семейства Linux.

**2. Краткие теоретические сведения**: см. главу 7, стр. 165. Пример кода: файл threads.c в разделе «Примеры программ из пособия», подраздел «Threads».

**3. Методические указания.**

3.1. Проект может быть реализован в виде консольного приложения в среде ОС Ubuntu, Fedora, CentOS или других средствами компилятора gcc версии не ниже 4.

3.2. Проект является модификацией лабораторной работы № 3, родительское приложение которой преобразуется в функцию главного потока, а дочернее – в функцию порождаемого потока, реализующую функциональность программы из лабораторной работы № 2 для обработки каждого из нескольких входных файлов.

3.3. Проект должен содержать функции порождения потока (в нужном количестве экземпляров), запуск в каждом порожденном потоке функции, реализующей функциональность из лабораторной работы № 2 с необходимым ей параметром, ожидания завершения каждого потока и получения его кода завершения.

3.4. Проект должен предусматривать синхронизацию потоков посредством мьютексов. Обмен данными между потоками может осуществляться через глобальный массив структур.

3.5. Проект должен предусматривать обработку исключительных ситуаций (недостаточное количество аргументов командной строки, невозможность захвата мьютекса, загрузки динамической библиотеки и импорта функции из нее).

3.6. Проект рекомендуется реализовать в 2 этапа:

3.6.1. Преобразование дочернего процесса лабораторной работы № 3 в функцию (далее оформляемую как функция порождаемого потока).

3.6.2. Преобразование родительского процесса лабораторной работы № 3 в функцию main (головного потока), содержащую все необходимые системные вызовы.

**4. Порядок выполнения работы.**

4.1. Переписать программу родительского процесса из лабораторной работы № 3, запускающую потоки, реализующие функциональ-

ность программы из лабораторной работы № 2. Количество потоков должно соответствовать числу входных файлов. Программа должна ждать завершения всех потоков и выводить на терминал коды их завершения.

4.2. Результатом работы программы являются выходные текстовые файлы, содержащие текст, обработанный согласно вариантам, и возвращаемые дочерними процессами значения – количество выполненных операций или –1 в случае ошибки.

4.3. При обработке исключительных ситуаций сообщения об ошибках выводятся на терминал.

4.4. Результатом выполнения лабораторной работы считается демонстрация работы программы с несколькими входными файлами и обработки исключительных ситуаций преподавателю.

**5. Варианты заданий**: см. лабораторную работу № 1 (стр. 214).

**6. Содержание отчета.**

6.1. Цель работы.

6.2. Вариант задания.

6.3. Листинг программы.

**7. Контрольные вопросы**: см. вопросы для самопроверки к главе 7 (стр. 187).

# Лабораторная работа № 8

# Сетевое взаимодействие процессов в Linux

**1. Цель работы**: получить навыки разработки сетевых приложений с использованием протоколов TCP и UDP на языке C в операционных системах семейства Linux.

**2. Краткие теоретические сведения**: см. главу 8, стр. 188.

**3. Методические указания.**

3.1. Проект может быть реализован в виде консольного приложения в среде ОС Ubuntu, Fedora, CentOS или других средствами компилятора gcc версии не ниже 4.

3.2. Проект является модификацией любой из лабораторных работ № 4–6, реализованных в клиент-серверной архитектуре.

3.3. Проект должен предусматривать обработку исключительных ситуаций (отсутствие или неверное количество входных параметров, ошибки системных вызовов, ошибки чтения и записи).

**4. Порядок выполнения работы.**

4.1.Модифицировать программы (серверную и клиентскую) из лабораторной работы № 4–6 для передачи данных через сокеты.

4.2. Результатом работы программы является выходной текстовый файл, содержащий текст, обработанный согласно вариантам, и возвращаемое значение – количество выполненных операций или –1 в случае ошибки.

4.3. При обработке исключительных ситуаций сообщения об ошибках выводятся на терминал.

4.4. Результатом выполнения лабораторной работы считается демонстрация работы программы и обработки исключительных ситуаций преподавателю.

**5. Варианты заданий**: см. лабораторную работу № 1 (стр. 214) Протокол передачи данных для нечетных вариантов: TCP, для четных: UDP.

**6. Содержание отчета.**

6.1. Цель работы.

6.2. Вариант задания.

6.3. Листинг программы.

**7. Контрольные вопросы**: см. вопросы для самопроверки к главе 8 (стр. 201).

# Курсовая (расчетно-графическая) работа

**1. Цель работы**: получить навыки создания и журналирования работы программ-демонов на языке С в операционных системах семейства Linux.

**2. Краткие теоретические сведения**: см. главу 9, стр. 202.

**3. Методические указания.**

3.1.Проект может быть реализован в виде консольного приложения в среде ОС Ubuntu, Fedora, CentOS или других средствами компилятора gcc версии не ниже 4.

3.2. Проект является модификацией серверных частей лабораторных работ № 4–8 (при условии, что лабораторная работа выполнялась с применением каналов FIFO), клиентская часть не изменяется.

3.3. Проект должен предусматривать обработку исключительных ситуаций (отсутствие или неверное количество входных параметров, ошибки открытия входного и/или выходного файла, ошибки чтения и записи).

3.4. Проект рекомендуется реализовать в 2 этапа.

3.4.1. Преобразование операций вывода (включая сообщения об ошибках) в серверном приложении в операции с журналом **`syslogd`**.

3.4.2. Преобразование серверного приложения в демона.

**4. Порядок выполнения работы.**

4.1. Модифицировать серверное приложение из лабораторной работы № 4–8, преобразуя операции вывода в консоль в операции с журналом **`syslogd`**.

4.2. Модифицировать серверное приложение в демона, продемонстрировать работоспособность программы преподавателю.

4.3. Результатом выполнения лабораторной работы считается демонстрация работы клиентского приложения с демоном и журнала работы демона.

**5. Варианты заданий**: см. лабораторную работу № 1 (стр. 214).

**6. Содержание отчета.**

6.1. Цель работы.

6.2. Вариант задания.

6.3. Листинги программ.

**7. Контрольные вопросы**: см. вопросы для самопроверки к главе 9 (стр. 211).

# Библиографический список


1. *Гунько А. В.* Программирование (в среде Windows) : учеб. пособие / А. В. Гунько. – Новосибирск : Изд-во НГТУ, 2019. – 155 с.

2. *Гунько А. В.* Системное программное обеспечение: конспект лекций : учеб. пособие / А. В. Гунько. – Новосибирск : Изд-во НГТУ, 2011. – 138 с.

3. *Гордеев А. В.* Системное программное обеспечение / А. В. Гордеев, А. Ю. Молчанов. – Санкт-Петербург : Питер, 2002. – 736 с.

4. *Стивенс У. Ричард* UNIX. Профессиональное программирование / У. Ричард Стивенс, Стивен А. Раго. – 3-е изд. – Санкт-Петербург : Питер, 2018. – 944 с.

5. *Керриск Майкл* Linux API. Исчерпывающее руководство / Майкл Керриск. – Санкт-Петербург : Питер, 2018. – 1248 с.

6. *Гунько А. В.* Системное программное обеспечение пособие / А. В. Гунько. – URL: http://gun.cs.nstu.ru/ssw (дата обращения: 16.03.2020).

7. *Стивенс У.* UNIX: взаимодействие процессов / У. Стивенс. – Санкт-Петербург : Питер, 2002. – 624 с.

8. Интерактивная система просмотра системных руководств (man-ов) – URL: https://www.opennet.ru/man.shtml (дата обращения: 16.03.2020).

9. *Стивенс У. Р.* UNIX: Разработка сетевых приложений / У. Р. Стивенс, Б. Феннер, Э. М. Рудофф. – Санкт-Петербург : Питер, 2007. – 1040 с.

# Оглавление

**Гунько Андрей Васильевич**

**СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ В СРЕДЕ LINUX**

**Учебное пособие**

Редактор *Е.Е. Татарникова*
Выпускающий редактор *И.П. Брованова*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *С.И. Ткачева*

Налоговая льгота – Общероссийский классификатор продукции
Издание соответствует коду 95 3000 ОК 005-93 (ОКП)

---

Подписано в печать 30.04.2020. Формат 60 × 84 1/16. Бумага офсетная. Тираж 25 экз.
Уч.-изд. л. 13,71. Печ. л. 14,75. Изд. № 67. Заказ № 577. Цена договорная

---

Отпечатано в типографии
Новосибирского государственного технического университета
630073, г. Новосибирск, пр. К. Маркса, 20