Lab127

Lab127 Lab0x7F Lab0177 Lab1111111b

Е. Д. Жиганов, А. П. Мощевикин

передача данных в компьютерных сетях

УЧЕБНОЕ ПОСОБИЕ

Ethernet + TCP/IP + ICMP + HTTP + SMTP

Федеральное агентство по образованию
Государственное образовательное учреждение
высшего профессионального образования
ПЕТРОЗАВОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Е. Д. Жиганов
А. П. Мощевикин

Передача данных в компьютерных сетях

Учебное пособие

Петрозаводск
Издательство ПетрГУ
2007

УДК 681.324
ББК 32.973.202
Ж68

Печатается по решению редакционно-издательского совета
Петрозаводского государственного университета

Рецензенты:

канд. т. н., ст. преподаватель	*Ю. В. Сидоров*
ст. преподаватель	*А. В. Соловьев*
CCNA сертифицированный инженер	*А. А. Корольков*

Жиганов, Е. Д.

Ж68 Передача данных в компьютерных сетях : учеб. пособие / Е. Д. Жиганов, А. П. Мощевикин. – Петрозаводск : Изд-во ПетрГУ, 2007. – 156 с.

ISBN 978-5-8021-0632-7

Учебное пособие предназначено для сопровождения лабораторного практикума по курсам, связанным с изучением сетевых технологий передачи данных и программированием сетевых интерфейсов. В издании приведены краткие сведения и справочные данные по некоторым сетевым технологиям, протоколам и утилитам, используемым в локальных и глобальных сетях, описаны способы создания сетевых приложений в Unix-подобных операционных системах; содержатся методические рекомендации и тексты заданий к лабораторным работам.
Пособие адресовано студентам физико-технического факультета, обучающимся по специальностям "Автоматизированные системы обработки информации и управления", "Информационно-измерительная техника и технологии", "Физическая электроника" и изучающим курсы "Сети ЭВМ и телекоммуникации" и "Сетевые технологии".

УДК 681.324
ББК 32.973.202

ISBN 978-5-8021-0632-7

Содержание

Предисловие

Учебное пособие "Передача данных в компьютерных сетях" предназначено для студентов специальностей АСОИУ, ИИТТ, ФЭ и др., изучающих курсы "Сети ЭВМ и телекоммуникации" и "Сетевые технологии". Первая часть пособия представляет собой краткие сведения и справочные данные по некоторым сетевым технологиям, протоколам и утилитам, используемым в локальных и глобальных сетях. Вторая часть посвящена описанию способов создания сетевых приложений в Unix-подобных операционных системах. Третья часть содержит методические рекомендации и задания к лабораторным работам. Пособие не заменяет материал вышеупомянутых курсов, а лишь дополняет их.

Издание подготовлено в рамках проекта "Научно-образовательный центр по фундаментальным проблемам приложений физики низкотемпературной плазмы" (RUX0-013-PZ-06), поддерживаемого Министерством образования и науки РФ, Американским фондом гражданских исследований и развития (CRDF) и Правительством Республики Карелия, а также программы Европейского Сообщества TACIS/Interreg.

EUREGIO KARELIA

Авторы выражают благодарность Л. Л. Пяхтину за сотрудничество при создании методических указаний к лабораторным работам № 5 и 6, Ю. В. Сидорову, А. В. Соловьеву и А. А. Королькову за рецензию, проверку текста и полезные рекомендации, а также И. М. Некрыловой за тщательную корректуру пособия.

ЧАСТЬ I

СЕТЕВЫЕ ТЕХНОЛОГИИ, ПРОТОКОЛЫ, УТИЛИТЫ

Глава 1.
Введение в концепцию OSI/RM, стек протоколов

В литературе по компьютерной и сетевой тематике слова "стандарт" и "протокол" упрощенно отражают принадлежность оборудования или программного обеспечения к тому или иному заранее определенному типу или классу. При этом задача классификации возложена на международные или национальные комитеты (консорциумы), которые фактически "законодательно" закрепляют пути развития техники и технологий. Как показывает опыт мирового экономического развития, на рынке выживают только компании, придерживающиеся этих законов, а выигрывает от этого именно потребитель. Единые стандарты порождают конкуренцию, облегчают изучение сходных по назначению продуктов технологического прогресса, позволяют заменять одни части и механизмы другими.

Простейшая вертикальная модель взаимодействия объектов в компьютерной архитектуре может быть описана так (сверху вниз):

- прикладное программное обеспечение (ПО) пользователя;
- операционная система;
- драйверы;
- аппаратура.

Каждое из этих понятий является самодостаточным и целостным, именно поэтому имеет право называться "уровнем". Инженер (программист) при создании того или иного продукта обычно работает на одном из них. Он либо пишет пользовательскую программу, которая использует стандартные (опять же заранее определенные) функции операционной системы (ОС), либо разрабатывает программную связку в виде драйвера, привязывающую вызовы операционной системы к конкретному аппаратному обеспечению, либо вообще создает достаточно интеллектуальное устройство, обычно состоящее из нескольких блоков, включающих, в том числе, и встроенное ПО ("прошивку").

Необходимо еще раз заострить внимание на том, что однородные продукты одного и того же уровня (или нескольких уровней) сохраняют свойство взаимозаменяемости. Так, например, можно сменить драйвер видеокарты на новый, написанный сторонним производителем, оставив неизменными саму карту и операционную систему, или установить другую программную оболочку-надстройку над ОС, например браузер, файловый менеджер, почтовый клиент и т. д.

1.1. Open System Interconnection Reference Model

Эталонная модель взаимодействия открытых систем (ЭМВОС, OSI/RM) предлагает несколько иную многоуровневую схему, именуемую стеком. Здесь и далее слово "стек" означает набор связанных между собой протоколов, процедур, стандартизованных правил, логически объединенных по вертикальному принципу. Данные передаются либо сверху вниз при их отправке в среду передачи, либо снизу вверх при получении программным обеспечением выского уровня.

Прикладной уровень
Уровень представления данных
Сеансовый уровень
Транспортный уровень
Сетевой уровень
Канальный уровень
Физический уровень

Пользователь почти всегда общается с компьютером или каким-нибудь другим телекоммуникационным устройством (например, сотовым телефоном) на прикладном уровне. Графический (или текстовый) интерфейс такого устройства обычно выполнен в диалоговом режиме и носит дружественный характер, интуитивно понятный человеку.

Типичное назначение уровня представления данных (представительского уровня) – видоизменить поток данных прикладного уровня. Это может быть просто форматирование текста, его перекодировка, сжатие и шифрация информации перед передачей ее в сеть, смена порядка байт (локальный/сетевой, см. соответствующий раздел). Для прикладного программиста услуги этого уровня могут быть вызваны соответствующими функциями обычно с передачей в качестве параметра указателя на строку (массив) с данными.

Для того чтобы подготовленные данные можно было передать удаленной стороне, необходимо обеспечить соответствующий канал связи. Все пять нижележащих уровней в той или иной мере этот канал и организуют.

Физический уровень описывает среду передачи (электрический кабель, оптоволокно, разъемы, частоту и способ модуляции для радиоволн, другие параметры).

Канальный уровень чаще всего ответственен за формирование кадров (четко выдержанных по размеру многобайтовых структур), содержащих систему адресации (например, **MAC**-адрес[1] источника, **MAC**-адрес назначения), поле управления и поле с данными более высокого уровня, сетевого. Канальный уровень разделен на два подуровня: нижний – **MAC** (Medium Access Control, управление доступом к среде) и верхний – **LLC** (Link Logical Control, логическое управление связью).

Сетевой уровень также имеет свою систему адресации (например, **IP**-адресация), поле управления и поле данных более высокого уровня, транспортного. Таким образом, в каждом кадре информация транспорт-

[1] **MAC** – Medium Access Control (управление доступом к среде); **MAC**-адрес канального уровня состоит из 6 байтов; существуют три типа **MAC**-адресов (в зависимости от значения самого старшего байта): уникальный, групповой рассылки и широковещательный; чаще всего этот адрес аппаратно закреплен за оборудованием, по его значению можно определить производителя сетевого устройства.

ного уровня инкапсулирована в поле данных сетевого уровня, а информация сетевого уровня – в поле данных канального уровня.

Было упомянуто, что и канальный, и сетевой уровни обычно имеют свои собственные системы адресации. Существование нескольких таких систем может показаться избыточным, но это не так.

Во-первых, благодаря сложной иерархической структуре сетевого адреса (деление всего адресного пространства на сети и подсети; особенно ярко это выражено для сетей **IPv6** с адресным пространством $\sim 2^{128}$ сетевых интерфейсов) резко снижаются требования к аппаратному обеспечению маршрутизаторов и упрощается их настройка (очень сложно установить правила перенаправления пакетов, не имея средств для логического или физического разделения оконечных устройств на группы). И во-вторых, становится возможным построение глобальных сетей на гетерогенной аппаратной основе (поверх разных устройств канального уровня). Так, операционную систему, поддерживающую стек **TCP/IP**, можно установить на компьютеры с сетевой платой стандарта Ethernet, аналоговым модемом, цифровым интерфейсом **ISDN**, платой коммутации стандарта **ATM** и т. д.

Таким образом, уровни ниже транспортного (сетевой, канальный, физический) предназначены для идентификации сетевых устройств и организации физического канала обмена информацией.

Необходимость включения транспортного и сеансового уровней как отдельных модулей в модель **OSI/RM** также не вызывает сомнений.

Представим себе ситуацию, когда по одному каналу связи (упрощенно, по одному проводу) общаются два оконечных компьютера, при этом на них попарно запущены несколько разнородных программ (например, на том и другом есть почтовые клиенты, программное обеспечение системы сигнализации, службы удаленного доступа и т. д.), посылающих друг другу информацию. При приеме пакетов на сетевом уровне происходит идентификация адреса назначения, и если этот адрес совпадает с адресом компьютера, принявшего пакет, это означает, что данные предназначались именно ему. Но как узнать, какому программному обеспечению на этом компьютере необходимо передать данные для обработки и визуализации (ведь почтовое сообщение нельзя отобразить программой, "не понимающей", что это такое)?

Выход находится в использовании дополнительных средств мультиплексирования (демультиплексирования) каналов связи на транспортном

уровне. Каждое приложение имеет свой уникальный номер (например, порт **TCP**, указываемый в заголовках транспортного уровня). Любая информация, приходящая на него, будет передана заранее определенному программистом или операционной системой обработчику. Скажем, информация-запрос, поступающая на порт 80, будет получена программой веб-сервера.

Кроме этого, одна из важнейших функций транспортного уровня – контролировать правильность приема потока данных на основе расчета контрольных сумм.

На сеансовом уровне лежит задача организации сеанса связи (по уже установленному "каналу" на четырех нижних уровнях модели **OSI/RM**); он отвечает за расставление контрольных точек, подтверждение приема данных, включает в себя процедуры корректного и некорректного завершения связи.

Обращаясь к стеку **TCP/IP**[2], действие процедур сеансового уровня можно продемонстрировать на примере установления **TCP**-соединения. Для этого сторона-инициатор должна отправить по определенному IP-адресу на определенный **TCP**-порт специальный пакет с выставленным флагом синхронизации **SYN** (synchronization). Удаленная сторона ответит на этот пакет другим пакетом с флагом **ACK** (acknowledgement), подтверждающим прием **SYN**. Сторона-инициатор снова известит удаленную сторону о том, что его **ACK** получен. Таким образом, две стороны взаимодействия "договорятся" об установлении канала связи, конечными точками (параметрами) которого являются два **IP**-адреса (информация сетевого уровня) и два **TCP**-порта (информация транспортного уровня), по одному на каждой стороне.

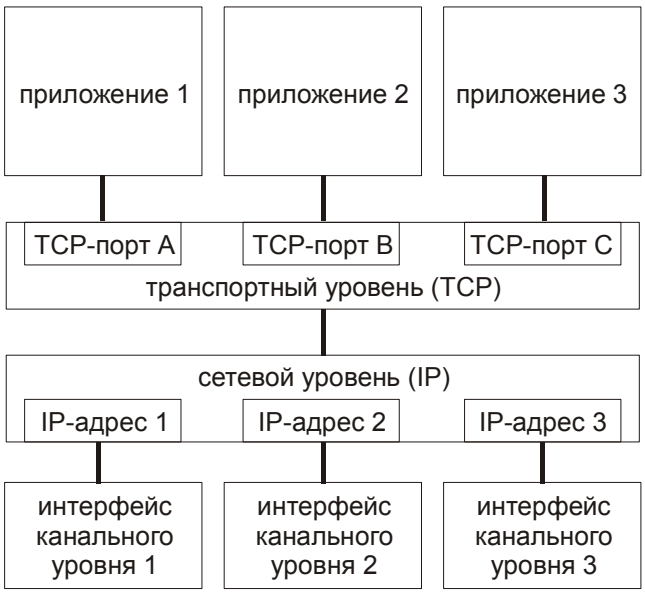
Повторяя все вышесказанное в терминах **TCP/IP**, можно описать ситуацию следующим образом.

Пусть в компьютере включены несколько интерфейсов канального (и физического) уровней, например аналоговый модем и две сетевые платы. За каждым из этих устройств может быть закреплен свой адрес сетевого уровня (в данном случае **IP**-адрес). Функции сетевого уровня по приему данных выполняет операционная система, она анализирует все полученные пакеты и определяет те, информацию из которых

---

[2] **TCP/IP** – Transmission Control Protocol / Internet Protocol (протокол управления передачей / межсетевой протокол); стек протоколов, на котором базируется глобальная сеть Internet.

необходимо передать наверх программным средствам транспортного уровня. Транспортный уровень также имеет свою систему адресации – **TCP**-порты. Из большого пространства портов (более 65 тысяч) операционная система прослушивает только те из них, что связаны с исполняемыми в данный момент приложениями.

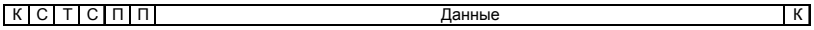


Таким образом, если поступающая информация была предназначена именно данному компьютеру (контроль осуществляется на канальном и сетевом уровне **OSI/RM**) и соответствует действующим (открытым) портам транспортного уровня, то она будет передана наверх и обработана программным средством прикладного уровня.

1.2. Структура сетевых пакетов

При формировании кадра, отправляемого в сеть, каждый уровень добавляет к байтам, поступающим с верхнего уровня, свои служебные данные в виде заголовков или трейлеров (конечных ограничителей).

Поэтому сформированный кадр выглядит следующим образом:



К – заголовок и трейлер канального уровня;
С – заголовок сетевого уровня;
Т – заголовок транспортного уровня;
С – заголовок сеансового уровня;
П – заголовок уровня представления данных;
П – заголовок прикладного уровня.

На стороне приемника при движении данных снизу вверх по стеку протоколов соответствующее программное обеспечение анализирует заголовок своего уровня и передает вложенные в него данные программным средствам вышестоящего уровня.

Контрольные вопросы к главе 1

1. Перечислите назначение и функциональные признаки всех семи уровней по модели **OSI/RM**.
2. Для какой цели существуют каждая из систем адресации на канальном, сетевом и прикладном (доменные имена) уровнях?
3. Что такое **TCP-**порт?
4. Опишите структуру сетевых пакетов.

Глава 2.
Технология Ethernet

Ethernet – технология (сетевая архитектура) локальных вычислительных сетей, описанная стандартами физического и канального уровней модели **OSI/RM**. При построении сети на коммутаторах и репитерах (повторителях, хабах) Ethernet строится по физической топологии "звезда"; логическая топология этой архитектуры вне зависимости от кабельной разводки всегда остается "шиной" (в случае использования **CSMA/CD** в качестве метода доступа к среде передачи)[3].

Скорость передачи данных определяется спецификацией и может равняться 10 Мбит/с, 100 Мбит/с (Fast Ethernet), 1 Гбит/с (Gigabit Ethernet), 10 Гбит/с (10 Gigabit Ethernet). Внутри каждой спецификации существует еще несколько подвидов (например, 100Base-TX, 100Base-FX для Fast Ethernet), характеризуемых разными видами подключения к среде передачи (оптоволокно, витая пара, коаксиальный кабель), а также методами кодирования сигнала и включением/выключением тех или иных коммуникационных опций.

Как уже было сказано, на канальном уровне все устройства имеют свой адрес, обычно определенный аппаратно. В технологии Ethernet в качестве адреса используется 6-байтовый идентификатор **MAC** (medium access control, например 00:00:C0:5E:83:0E).

Различают широковещательные (broadcast), уникальные (unicast) **MAC**-адреса и **MAC**-адреса групповой рассылки (multicast). Первый состоит из 1 во всех 48 разрядах (FF:FF:FF:FF:FF:FF), у второго самый левый (старший) бит всегда 0. **MAC**-адрес групповой рассылки обязательно

---

[3] Физическая топология – вид кабельной разводки и подключения оконечных узлов к коммутационному оборудованию.
Логическая топология – метод и способ доступа к среде передачи.
Звезда характеризует способ подключения, при котором каждое оконечное оборудование подключается к одному выделенному коммутационному устройству посредством отдельных кабельных соединений.
Шина характеризует такой способ подключения устройств, при котором можно выделить разделяемую всеми среду передачи (например, коаксиальный кабель в технологии "тонкий Ethernet 10Base-2", если говорить о физической топологии) и/или способ взаимодействия, при котором оконечные равноправные устройства "прослушивают" сеть, ожидая момент разрешения начала передачи, когда среда становится свободной.

содержит 1 в самой старшем бите самого старшего байта. Остальные биты могут принимать любые значения.

В качестве алгоритма доступа к среде передачи используется метод **CSMA/CD** (Carrier Sense Multiple Access with Collision Detection, множественный доступ с прослушиванием несущей и обнаружением коллизий).

Все станции, находящиеся в пределах одного коммутационного узла, прослушивают сеть (логическая топология "шина") и равноправны по отношению к моменту начала передачи, которая может начинаться только при отсутствии сигнала других станций. Момент начала передачи ничем не регламентирован, такой метод доступа относится к категории недетерминированных.

2.1. Алгоритм CSMA/CD

1. Перед началом передачи станция должна определить, "свободна" ли среда передачи (прослушивание несущей).

2. Если чужой сигнал в среде не обнаружен, станция может начать передачу.

3. Во время начала передачи станция также прослушивает сеть на предмет коллизии (искажение формы сигналов из-за их наложения), которая может произойти вследствие множественного доступа и почти одновременного начала передачи двумя и более станциями. Если она обнаружена, станция посылает в сеть специальный "jam"-сигнал, облегчающий обнаружение коллизии другим станциям, а также прекращает отсылку данных и переходит в режим ожидания.

4. Период ожидания определяется случайным образом, и его длительность также зависит от количества последовательно произошедших коллизий.

5. После выхода из режима ожидания станция снова может начинать передачу (переход на пункт 1).

Метод доступа к среде в сетях Ethernet (CSMA/CD) носит конкурентный характер. Станции прослушивают среду, и если она свободна, имеют право начать передачу. Из-за неопределенности этого момента и достаточно большой длины кабельной системы в среде могут возникать коллизии (то есть "столкновения" сигналов). Это приводит

к необходимости повторной передачи через некоторый, случайным образом выбранный, интервал времени. Коллизии уменьшают общую пропускную способность сети.

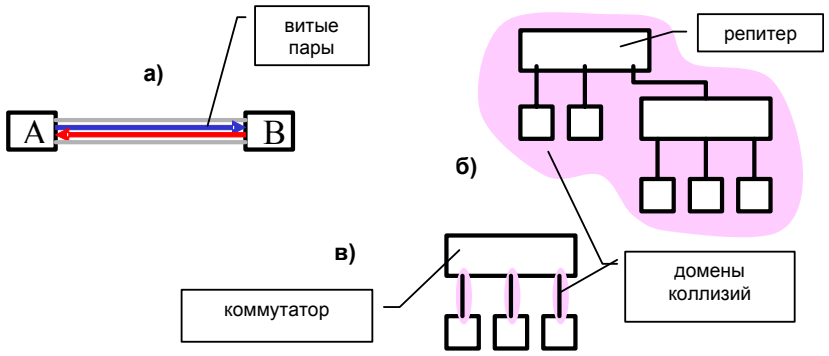
Домен коллизий – это объединенная часть кабельной системы, станций и другого коммуникационного оборудования, в которой возможно образование коллизий, **или** часть сети Ethernet, в которой нет буферизирующих кадры устройств (например, коммутаторов с проверкой корректности полученного кадра), **или** множество всех станций сети, одновременная передача любой пары из которых приводит к коллизии.

Возможны следующие случаи подключения сетевых устройств (см. рис.):

а) коллизий не существует (сетевые карты работают в дуплексном режиме);

б) если сеть построена на репитерах, то домен коллизий включает в себя всю кабельную систему (сетевые карты работают в режиме полудуплекса);

в) домен коллизий ограничен кабелем от сетевой карты до коммутатора (сетевые карты работают в полудуплексном режиме).



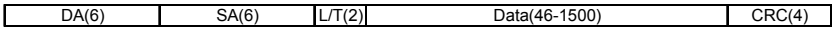
Отсюда становится понятным, что в полнодуплексном режиме при использовании технологии Fast Ethernet в случае а) общая пропускная способность между двумя компьютерами будет 200 Мбит/сек. В случае использования репитеров (повторителей, хабов) общая пропускная

способность понижается по мере подключения новых компьютеров в сеть (обработка ситуации с коллизиями отбирает часть времени, которое могло быть затрачено на передачу полезного трафика).

Самой выгодной с точки зрения увеличения общей пропускной способности является проектирование и монтаж сетей Ethernet на коммутаторах или других коммуникационных устройствах, умеющих разделять порты и буферизировать кадры.

2.2. Формат кадра Ethernet

Существует 4 различных вида кадров Ethernet, их общий вид представлен ниже.



- DA – адрес назначения (Destination Address, 6 байтов);
- SA – адрес источника (Source Address, 6 байтов);
- L/T – длина или тип кадра (Length/Type, 2 байта);
- Data – данные верхнего уровня (например, IP-уровня, 46–1500 байтов);
- CRC – поле контрольной суммы (Cyclical Redundancy Check, 4 байта).

Необходимо также отметить, что перед каждым кадром станция-отправитель добавляет преамбулу и начальный ограничитель кадра (8 байтов). Кроме того, все станции должны выдерживать межкадровые промежутки в 96 тактов (например, 9.6 мкс при битовой скорости 10 Мбит/с).

2.3. Promiscuous mode (режим прослушивания сети)

В обычном режиме функционирования сетевого интерфейса при получении кадра данные (46–1500 байтов) будут переданы обработчику верхнего уровня только в случаях, если адрес назначения, установленный в поле DA, широковещательный либо он совпадет с уникальным **MAC**-адресом принимающей станции.

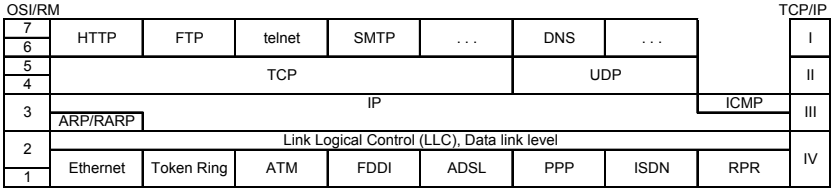
Однако каждый адаптер Ethernet может быть переведен в режим, в котором будут обрабатываться все кадры, поступающие из среды передачи. На английском языке такой режим носит название "promiscuous", что переводится как "безразличный" или "неразборчивый". Этим свойством сетевых адаптеров можно пользоваться, например, для создания программных анализаторов сетевого трафика (**tcpdump**, см. главу с описанием сетевых утилит).

Контрольные вопросы к главе 2

1. Опишите метод доступа к среде передачи **CSMA/CD**, используемый в Ethernet.
2. Что такое **MAC**-адрес? Укажите уровень по модели **OSI/RM**, в рамках которого уместно упоминать **MAC**-адрес.
3. В чем разница между физической и логической топологиями посроения сетей?
4. Рассчитайте полезную пропускную способность сети Fast Ethernet (100 Мбит/с), учитывая в качестве накладных расходов межкадровые промежутки, преамбулу кадра и заголовки канального уровня.
5. В каком случае при поступлении кадра с физического уровня станция будет "изучать" поле <данные> канального уровня?
6. Что такое promiscuous режим?
7. Укажите максимальное количество возможных уникальных (unicast) MAC-адресов.

Глава 3.
Протоколы стека TCP/IP и прикладного уровня

Стек **TCP**/**IP** (Transmission Control Protocol / Internet Protocol, протокол управления передачей / межсетевой протокол) в отличие от **OSI**/**RM** содержит всего 4 уровня: I – прикладной, II – транспортный, III – межсетевой, IV – физический (физического интерфейса). Все они в той или иной степени соответствуют уровням идеальной модели, то есть выполняют похожие функции.



Уровень физического интерфейса в **TCP**/**IP** не регламентирован, в качестве него может выступать одна из сетевых архитектур, обеспечивающая передачу/прием кадров канального уровня (**Ethernet**, Token Ring, **ATM**, **FDDI**, **ADSL**, **PPP**, **ISDN**, **RPR** и т. д.). Верхний подуровень канального (Data link) уровня, именуемый **LLC**, обеспечивает связку между IV и III уровнями стека **TCP**/**IP**. Следует отметить, что во многих случаях **IP**-уровень может быть расположен непосредственно над уровнем физического интерфейса (минуя подуровень **LLC**).

В качестве основы межсетевого уровня следует выделить собственно **IP**-уровень, отвечающий за адресацию и процесс маршрутизации в глобальных сетях.

ARP/**RARP**[4] – протокол, заголовки и данные которого инкапсулированы непосредственно в кадр канального уровня, а не в **IP**-датаграмму[5], предназначен для взаимосвязи адресации канального (чаще всего **MAC**-адресация) и сетевого уровней (**IP**-адресация). **ICMP**[6] инкапсулирован в

---

[4] **ARP/RARP** – Address Resolution Protocol / Reversed ARP (протокол распознавания адреса / протокол обратного распознавания адреса).

[5] Датаграмма – блок данных, в заголовочной части которого указаны адреса сетевого уровня источника и приемника.

[6] **ICMP** – Internet Control Message Protocol (межсетевой протокол управляющих сообщений).

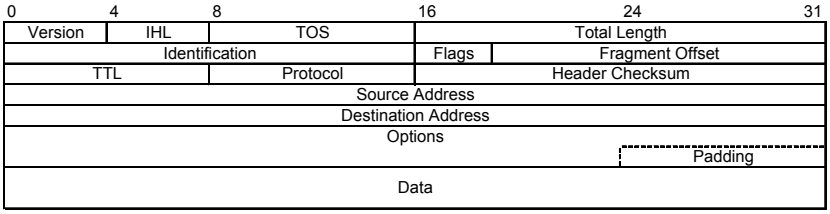
IP, но не содержит функций транспортного уровня, поэтому изображен на сетевом уровня, хоть и является по сути протоколом прикладного уровня.

Транспортный уровень представлен двумя способами организации связи: с гарантированной (**TCP**) и негарантированной (**UDP**[7]) доставкой данных.

Протоколы прикладного уровня и соответствующие приложения используют либо **TCP**, либо **UDP** для передачи информации между двумя сторонами сетевого взаимодействия.

3.1. Межсетевой протокол IP (Internet Protocol)

Протокол **IP** – это протокол сетевого уровня по модели **OSI/RM**. Формат **IP-**датаграммы представлен ниже; каждая строчка содержит 32 бита (4 байта).



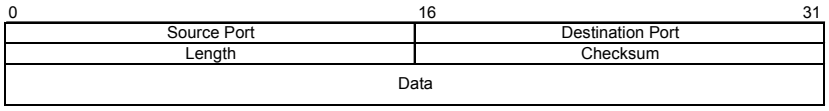
- Version – версия **IP**-протокола (для **IPv4** в это поле заносится 4, что соответствует в бинарном виде 0100 в старшем полубайте первого октета датаграммы);
- IHL – длина заголовка в 32-битовых словах (Internet Header Length, 4 бита; в случае **IPv4** и длины заголовка в 32-битовых словах, равной 5, в первом байте **IP-**датаграммы содержится 0x45);
- TOS – тип сервиса (Type of Service, 1 байт);
- TL – полная длина IP-датаграммы в байтах (Total Length, 2 байта);
- Identification – 16-битовое число, одинаковое для всех фрагментов одной датаграммы (2 байта);
- Flags – флаги (DF – не фрагментировать, MF – еще фрагменты, 3 бита, один флаг зарезервирован);

[7] **UDP** – User Datagram Protocol (протокол пользовательских датаграмм).

- Fragment Offset – смещение фрагмента внутри датаграммы в 8 байтовых единицах (13 битов);
- TTL – время жизни IP-датаграммы (Time To Live, значение в этом поле уменьшается на 1 при прохождении через очередной маршрутизатор, 1 байт);
- Protocol – протокол верхнего уровня (например, 0x06 для **TCP**, 0x11 для **UDP**, 0x01 для **ICMP**, 1 байт);
- Header Checksum – контрольная сумма заголовков IP-датаграммы (2 байта);
- Source Address – **IP-**адрес источника (4 байта);
- Destination Address – **IP-**адрес назначения (4 байта);
- Options – опции (если есть);
- Padding – байты выравнивания поля опций до 32-битового слова;
- Data – данные верхнего уровня.

3.2. Протокол UDP (User Datagram Protocol)

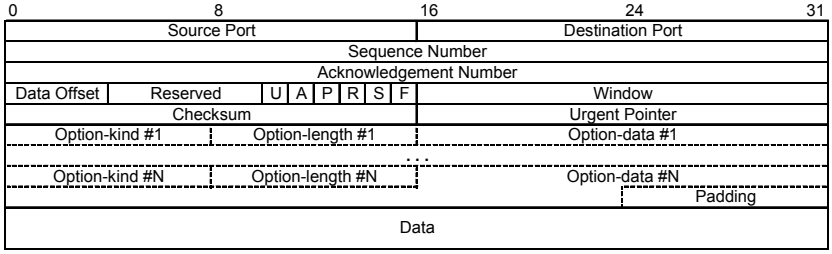
Протокол передачи пользовательских датаграмм **UDP** – протокол транспортного уровня по модели **OSI/RM**. Формат **UDP-**датаграммы представлен ниже.



- Source Port – порт источника (2 байта);
- Destination Port – порт назначения (2 байта);
- Length – полная длина **UDP-**пакета (2 байта);
- Checksum – контрольная сумма (2 байта);
- Data – данные, инкапсулированные в **UDP-**пакет.

3.3. Протокол TCP (Transmission Control Protocol)

Протокол управления передачей **TCP** – протокол транспортного и сеансового уровней по модели **OSI/RM**. Формат **TCP-**сегмента представлен ниже.



- Source Port – порт источника (2 байта);
- Destination Port – порт назначения (2 байта);
- Sequence Number – номер последовательности (4 байта);
- Acknowledgement Number – номер подтверждения (4 байта), оба этих числа используются для контроля правильности приема переданных данных;
- Data Offset – длина **TCP-**заголовков в 32-битовых словах (4 бита);
- однобитовые **TCP-**флаги (U, A, P, R, S, F);
- Window – размер окна в байтах (2 байта);
- Checksum – контрольная сумма **TCP-**сегмента и псевдозаголовков (2 байта);
- Urgent pointer – указатель срочности (2 байта);
- Option-kind, Option-length и Option-data – тип (1 байт), длина (1 байт) и данные опций **TCP**;
- Padding – поле заполнения до 32-битового слова;
- Data – данные верхнего уровня, инкапсулированные в **TCP-**сегмент.

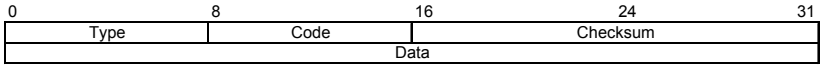
TCP-флаги:

- URG (urgent) – флаг срочности, указывает на использование поля Urgent Pointer (например, при нажатии Ctrl-C в сеансе telnet или при передаче файла в поле Urgent Pointer записывается смещение передаваемых в этом сегменте данных относительно номера последовательности);
- ACK (acknowledgement) – флаг подтверждения, указывает на подтверждение приема данных и необходимость использования Acknowledgement Number;
- PSH (push) – указатель немедленной передачи инкапсулированных данных приложению на стороне получателя (например, в сеансе

telnet для отправки сегментов, содержащих отдельные буквы, при побайтовой отсылке на сервер);
- RST (reset) – флаг сброса соединения;
- SYN (synchronization) – флаг инициирования соединения, указывает на необходимость использования Sequence Number;
- FIN (finish) – флаг запроса закрытия соединения.

3.4. Протокол ICMP (Internet Control Message Protocol)

ICMP (протокол управляющих сообщений в сети Internet) инкапсулирован в **IP**-датаграмму (при этом поле Protocol в **IP**-заголовках содержит 0x01).

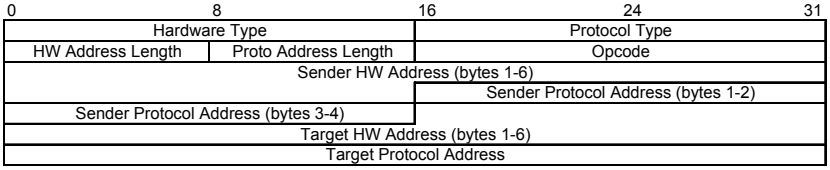


- Type – тип **ICMP**-сообщения (1 байт);
- Code – код сообщения (зависит от значения поля Type, 1 байт);
- Checksum – контрольная сумма;
- Data – данные.

3.5. Протоколы ARP и RARP (Address Resolution Protocol и Reversed ARP)

Протоколы **ARP** (распознавания **MAC**-адреса по известному **IP-**адресу) и **RARP** (распознавания **IP**-адреса по известному **MAC**-адресу) связывают канальный и сетевой уровни в модели **OSI/RM**. Оба протокола носят широковещательный характер, поэтому используются только в локальных сетях (в пределах одного сегмента сетевой архитектуры канального уровня, что называется "до первого маршрутизатора").

Необходимо обратить внимание, что **ARP** и **RARP** организуют взаимосвязь не только для **Ethernet** – **IP**, но и для других сетевых архитектур и протоколов сетевого уровня; формат **ARP** и **RARP** пакетов представлен ниже.

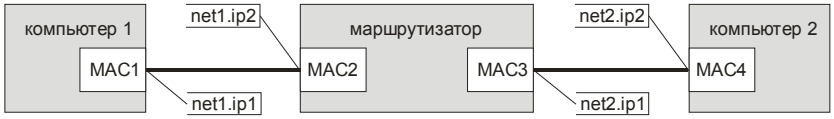


- Hardware Type – тип сетевой архитектуры (канальный уровень, 0x0001 для 10 Mb Ethernet);
- Protocol Type – идентификатор протокола сетевого уровня (0×0800 для **IP**);
- HW Address Length – длина адреса канального уровня (0×06 для **MAC-**адреса);
- Protocol Address Length – длина адреса сетевого уровня (0×04 для **IP-**адреса);
- Opcode – код операции (1 – **ARP-**запрос, 2 – **ARP-**ответ, 3 – **RARP-**запрос, 4 – **RARP-**ответ);
- Sender и Target HW Address и Protocol Address – адреса канального и сетевого уровней источника и приемника соответственно.

Часть полей может быть пустой. Так, при формировании **ARP-**запроса поле Target HW Address остается незаполненным. Станция, имеющая такую информацию, сформирует **ARP-**ответ, в котором укажет искомый адрес канального уровня.

3.6. Процесс отправки, перенаправления и получения датаграмм

Процесс отправки, перенаправления и получения датаграммы сетевого уровня можно рассмотреть на примере приведенного рисунка.



На схеме слева и справа изображены два компьютера, у каждого есть интерфейс **канального уровня** (например, сетевой адаптер стандарта

Ethernet) с уникальными адресами MAC1 и MAC4 соответственно. Между ними находится маршрутизатор (устройство **сетевого уровня** по модели **OSI/RM**), в задачу которого входит перенаправление датаграмм с одного из физических интерфейсов (MAC2, MAC3) на другой. Основанием для перенаправления является соответствие сетевого адреса назначения, считанного из заголовков сетевого уровня датаграммы, сети назначения, указанной в таблице маршрутизации. При этом компьютер 1 и левый физический интерфейс маршрутизатора имеют разные адреса сетевого уровня и входят в одну сеть net1, а компьютер 2 и правый интерфейс маршрутизатора – в сеть net2.

Итак, если у компьютера 1 возникает необходимость отправить датаграмму сетевого уровня компьютеру 2, то ему необходимо ее подготовить, инкапсулировать в кадр канального уровня и отправить его в среду передачи через свой единственный физический интерфейс. Во время подготовки датаграммы отправителю становятся известны адреса источника net1.ip1 (свой) и получателя net2.ip2 на сетевом уровне (например, удаленный адрес сетевого уровня можно узнать, воспользовавшись службой **DNS**). Сформированная датаграмма записывается в поле данных кадра канального уровня. Свой адрес канального уровня MAC1 отправителю, понятно, известен. Драйвер сетевой платы записывает его в поле SA (Source Address). Проблема возникает с тем, что писать в поле DA (Destination Address) и куда отправлять кадр с данными. Ответ очевиден: на маршрутизатор по умолчанию, коим является интерфейс сетевого уровня с адресом net1.ip2. **MAC**-адрес этого сетевого интерфейса можно узнать, воспользовавшись службой **ARP** (Address Resolution Protocol). Компьютер 1 формирует широковещательный на канальном уровне **ARP**-запрос (только в пределах левого сегмента, маршрутизатор не пропустит сквозь себя широковещательные кадры канального уровня) для того, чтобы по известному адресу сетевого уровня (net1.ip2) узнать его **MAC**-адрес.

После того, как SA и DA будут известны, кадр с инкапсулированной датаграммой отправляется маршрутизатору.

Маршрутизатор получает кадр с левого (на рисунке) интерфейса, вскрывает его и находит датаграмму сетевого уровня, в которой указан адрес назначения net2.ip2. По своей таблице маршрутизации, пользуясь маской сетевого адреса, он определяет, что датаграмму необходимо переслать на интерфейс с **MAC**-адресом MAC3. Поскольку компьютер 2 включен логически в сеть (на сетевом уровне модели **OSI/RM**), непосредственно подсоединенной к маршрутизатору, последний

сформирует **ARP-**запрос на адрес назначения net2.ip2. В ответ ему придет информация о MAC4. Он сформирует новый кадр канального уровня, вложив в него исходную неизмененную датаграмму.

Компьютер 2 получит новый кадр, вскроет его, проанализирует содержимое заголовков сетевого уровня и передаст содержимое датаграммы наверх по стеку.

3.7. Установление и разрыв TCP-соединения

Для того чтобы установить **TCP-**соединение, необходимо проделать следующие действия.

1. Сторона-инициатор соединения (клиент) отправляет SYN-сегмент (выставленный бит SYN в поле флагов заголовков TCP), указывая номер порта сервера, к которому клиент хочет подсоединиться, и исходный номер последовательности клиента (поле Sequence Number в заголовках **TCP**).
2. Сервер отвечает сегментом SYN, содержащим свой исходный номер последовательности сервера вместе с выставленным флагом SYN. Также он выставляет флаг ACK и заполняет поле "номер подтверждения" (Acknowledgement number), вставляя в него полученный от клиента номер последовательности + 1.
3. Клиент должен подтвердить приход SYN-сегмента от сервера с использованием ACK-флага и нового значения в поле подтверждения (полученный от сервера номер последовательности +1).

Передачи таких трех сегментов достаточно для установления соединения (часто этот процесс называется трехразовым рукопожатием, three-way handshaking). После этого между сторонами возможен двусторонний обмен данными по установившемуся соединению.

При одностороннем закрытии соединения сторона-инициатор закрытия должна послать по установленному соединению FIN-сегмент (выставленный флаг FIN), а также получить ACK-ответ от удаленной стороны с уведомлением о получении FIN-сегмента.

После этого соединение с возможностью двустороннего обмена данными переходит в однонаправленное состояние (одна сторона закрыла соединение, вторая, активная, поддерживает его открытым).

Для того чтобы полностью закрыть соединение, активная сторона должна сформировать FIN-сегмент и получить на него подтверждение.

Таким образом, при установлении соединения двум сторонам необходимо послать и принять 3 сегмента, при его разрыве – 4.

3.8. Hyper Text Transfer Protocol (HTTP)

Под аббревиатурой **HTTP** (протокол доставки гипертекстовых документов) понимается протокол, обеспечивающий функционирование Сети и являющийся его базовой транспортной подсистемой. На данный момент все средства просмотра страниц в Internet должны соответствовать спецификации HTTP/1.1, представленной в июне 1999 г. (RFC 2616).

Любой стандарт, связанный с Internet, публикуется в виде RFC-документов (Request for Comments, запрос на рецензию и комментарии). Основным разработчиком алгоритмов и процедурных правил клиент-серверного Web-взаимодействия следует признать Тима Бернерс-Ли (T. Berners-Lee), участвовавшего в создании всех версий протокола HTTP, начиная с HTTP/0.9 (1991 г.), HTTP/1.0 (окончательно утвержденного в RFC 1945 в мае 1996 г.) и заканчивая HTTP/1.1 (RFC 2068 и RFC 2616). Все упомянутые RFC-документы можно найти по адресу: http://www.ietf.org/ ("штаб-квартира" WWW).

Схема запроса и выдачи документов в среде Internet выглядит следующим образом. Клиентское программное обеспечение (Internet Explorer, другой браузер) посылает **HTTP**-запрос веб-серверу с указанием адреса конкретного документа, который потребовался человеку. Веб-сервер анализирует запрос и отсылает запрашиваемый документ. Причем это взаимодействие прозрачно для пользователя, он может наблюдать за процессом, лишь ориентируясь на строку состояния браузера.

Чаще всего страничка начнет появляться на экране монитора только после того, как полностью загрузится в память компьютера (это связано с особенностями отображения браузерами содержимого веб-страниц).

Протокол **HTTP** описывает порядок действий клиентского программного обеспечения в момент запроса им гипертекстовых документов и другой информации у серверов, а также отправку ответов этими серверами. Ранние версии протокола позволяли строить запрос в упрощенной форме, тогда как HTTP/1.1 требует, чтобы были явно

указаны несколько полей внутри него, например обязательное поле host. Браузеры могут заполнять специальные поля в заголовке **HTTP**-запроса для того, чтобы сервер присылал им документ лишь тогда, когда он устарел. Кроме того, они управляют кэшированием информации в промежуточных прокси-серверах, ставят указания о возможности архивирования содержимого ответа и кодировке символов. Также правилом "хорошего тона" можно считать отсылку серверу адреса ресурса, являющегося источником данного запроса, и дополнительной текстовой информации, называемой cookies (файл персонализации). Последняя возможность активно используется Web-мастерами для учета посещаемости их ресурсов, возможности корректировки внешнего вида сайта, сохранения определенных настроек пользователя и т. д.

Вот так, например, выглядит **HTTP**-запрос, посылаемый браузером открытым текстом по предварительно установленному **TCP**-каналу:

```
GET /index.shtml HTTP/1.1
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows NT;
DigExt)
Host: thermo.karelia.ru
Connection: Keep-Alive
Cookie: voted=21; uid=45639
```

В нем запрашивается index.shtml – главная страница сервера thermo.karelia.ru. При этом указываются:

- версия HTTP;
- возможность gzip- и deflate-компрессии;
- сигнатура браузера, сформировавшего запрос (Mozilla/4.0 (compatible; MSIE 5.0; Windows NT; DigExt));
- имя Web-сервера (thermo.karelia.ru);
- правило поддержания общения между браузером и Web-сервером (постоянное соединение);
- полученные ранее этим пользователем файлы персонализации (voted со значением "21" и uid со значением "45639").

Web-сервер (thermo.karelia.ru) обязан как-то отреагировать на такую информацию. Он либо пришлет требуемый документ, либо сформирует пакет, содержащий код ошибки. Если этого не произойдет, то через некоторое время клиентская сторона закроет соединение и отобразит пользователю сообщение о недоступности хоста.

Для того чтобы сформировать веб-серверу простейший (пусть даже некорректный) запрос, достаточно создать tcp-соединение с 80-м портом сервера (чаще всего именно этому порту приписан сервис httpd) и послать ему следующую команду:

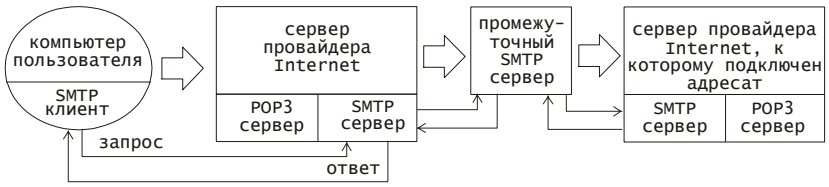
```
GET / HTTP/1.1
и нажать два раза Enter (<CRLF><CRLF>)
```

В ответ сервер, скорее всего, выдаст сообщение об ошибке.

3.9. Simple Mail Transfer Protocol (SMTP)

В качестве транспортного средства доставки сообщений в Internet выступает SMTP (simple mail transfer protocol, простой протокол передачи почты), который был стандартизован в виде RFC 821 (Request For Comments).

Упрощенно схема взаимодействия представлена на следующем рисунке (объемными стрелками показано направление движения почтовых сообщений).



Отправка почты по протоколу SMTP

Со стороны пользователя обычно одна и та же программа выступает в роли и POP3-клиента и SMTP-клиента отправителя. Наиболее распространенными на данный момент являются MS Outlook, The Bat, Netscape Messenger, Eudora, Pegasus mail, Mutt, Pine и др. При нажатии в них на кнопочку "отправить" происходит формирование очереди сообщений (если посылается не одно письмо) и установление двустороннего сеанса общения с SMTP-сервером провайдера. На схеме представлено так, что у пользователя есть клиентское ПО, а у провайдера – серверная часть приложения. На самом деле это немного не так. Протокол SMTP делает возможным смену сторон даже в ходе одного сеанса. Условно принято считать клиентом ту сторону, которая начина-

ет взаимодействие и хочет отослать почту, а сервером – ту, что принимает запросы. После того, как клиент посылает серверу несколько служебных команд и получает положительные ответы на них, он отправляет SMTP-серверу собственно тело сообщения. SMTP-сервер получает сообщение, вносит в него дополнительные заголовки, указывающие на то, что он обработал данное послание, устанавливает связь со следующим SMTP-сервером по пути следования письма. Общение между любыми SMTP-серверами происходит по той же схеме. Инициирует переговоры клиент, сервер на них отвечает, а затем получает корреспонденцию и "ставит штампик" в теле письма (в его заголовочной части). Все это очень напоминает обычную бумажную почту, где работу по сортировке и отправке почты выполняют люди.

Если на каком-нибудь этапе передачи SMTP-клиент обнаружит невозможность подключиться к следующему серверу (например, компьютер отправили на профилактику или аппаратура связи вышла из строя), он будет пытаться отправить сообщение через некоторое время – 1 час, 4 часа, день и т. д., до 4 суток в общем случае. Причем временные отрезки между попытками, как правило, зависят от настроек программы-пересыльщика почты. Одновременно такой сервер должен уведомить отправителя сообщения о невозможности доставить почту, послав ему стандартное письмо "Failed delivery" (доставка невозможна) и рассказав о графике дальнейших попыток по продвижению исходного сообщения. Если канал связи не восстановится за указанный большой промежуток времени (например, 4 дня), посланная информация будет считаться утерянной.

Как только почта достигнет конечного пункта (SMTP-сервера адресата сообщения), она будет сложена в почтовый ящик абонента, который всегда сможет в удобное для него время изъять ее по протоколам POP3 или IMAP в зависимости от того, какой из них поддерживается провайдером.

Анализируя "штампики" в заголовках полученного письма, можно узнать, какими путями оно путешествовало, как долго длился сам путь, как называлась почтовая программа отправителя и многое другое. Получить эту информацию можно в "Свойствах письма", кликнув правой кнопкой мыши на самом письме в MS Outlook, нажав Ctrl + Shift + H в The Bat или совершив нечто подобное в других почтовых клиентах.

```
Received: from mx10.mail.ru (mx10.mail.ru [194.67.57.20])
    by dfe3300.karelia.ru (8.9.0/8.9.0) with ESMTP id JAA02601
    for <somebody@dfe3300.karelia.ru>;
    Thu, 18 Apr 2002 09:19:13 +0400 (????)
Received: from f5.int ([10.0.0.57] helo=f5.mail.ru)
    by mx10.mail.ru with esmtp (Exim MX.A)
    id 16y46p-0002ox-00
    for somebody@dfe3300.karelia.ru; Thu, 18 Apr 2002 09:05:27
+0400
Received: from mail by f5.mail.ru with local (Exim FE.5)
    id 16y46o-000CfY-00
    for somebody@dfe3300.karelia.ru; Thu, 18 Apr 2002 09:05:26
+0400
Received: from [213.59.200.7] by win.mail.ru with HTTP;
    Thu, 18 Apr 2002 09:05:26 +0400
From: "Testing" <testing@mail.ru>
To: somebody@dfe3300.karelia.ru
Subject: For testing purposes only
Mime-Version: 1.0
X-Mailer: mPOP Web-Mail 2.19
X-Originating-IP: [213.59.200.7]
Date: Thu, 18 Apr 2002 09:05:26 +0400
Reply-To: "Testing" <testing@mail.ru>
Content-Type: text/plain; charset=koi8-r
Content-Transfer-Encoding: 8bit
Message-Id: <E16y46o-000CfY-00@f5.mail.ru>
X-UIDL: 74fb663e2be8352b3a0b88ca08030c1e


Тестовое сообщение.
```

Данный пример показывает четкую структуру письма. Тело почтового сообщения состоит из заголовочной части и текста (в данном случае слова "Тестовое сообщение") плюс вложения. До слова "From:" информация заносится промежуточными SMTP-серверами, каждый из которых добавляет свои данные в самое начало, тем самым постоянно увеличивая количество передаваемых далее байтов. Средняя заголовочная часть (начиная с "From:" и заканчивая "X-UIDL") почти полностью формируется почтовым агентом отправителя. Исключение составляет, например, поле "Message-Id:", которое было сформировано первым по пути следования SMTP-сервером – f5.mail.ru. Некоторые поля являются стандартизованными и желательными для употребления, например From, To, Subject, Reply-To… А некоторые являются дополнительными по отношению к описанным в стандарте текстовых сообщений в среде

Internet (RFC 822) и необязательными. Такие поля начинаются с "X-" и служат, например, для идентификации самого почтового клиента.

Строки `X-Mailer: mPOP Web-Mail 2.19` и `Received: from [213.59.200.7] by win.mail.ru with HTTP;` говорят о том, что письмо было отослано с помощью браузера с использованием протокола HTTP (hypertext transfer protocol). На пути от win.mail.ru к dfe3300.karelia.ru встретилось два транзитных SMTP-сервера: f5.mail.ru и mx10.mail.ru. Каждый из них поставил отметку о себе в заголовочную часть сообщения. Наличие нескольких почтовых серверов в домене mail.ru свидетельствует о масштабности данного проекта и особых правилах маршрутизации почты внутри него.

Составляя почтовое сообщение, пользователь должен заполнить поле "To", куда заносится адрес назначения (запись "From" подставится в заголовки письма автоматически). В Internet существуют правила хорошего тона, которые рекомендуют также указывать тему сообщения (поле "Subject").

Если необходимо разослать почту по нескольким адресам "в открытую", необходимо воспользоваться полем "CC" (carbon copy) или дописать их через запятую в поле "To". Слова "в открытую" означают, что человек, получивший такое письмо и просмотревший его заголовки, может определить, кому еще оно было послано в качестве копии. Если вдруг возникнет необходимость доставить один и тот же текст нескольким людям, но так, чтобы каждый из них не знал, кому еще адресовано это письмо, придется воспользоваться полем "BCC" (blind carbon copy – слепая копия). В этом случае все адресаты, указанные в поле "BCC" через запятую, получат сообщение, как если бы оно предназначалось только им. Эту работу по "размножению" почты выполняют незаметно для глаз пользователя почтовый клиент пользователя и первый по пути следования SMTP-агент. Они анализируют заголовочную часть письма, и если обнаруживают заполненные поля "CC", "BCC", то действуют в соответствии со своими алгоритмами обработки почтовых сообщений.

Теперь более подробно рассмотрим клиент-серверное взаимодействие по протоколу SMTP. Программа пользователя, выбрав для связи соответствующий почтовый сервер, устанавливает с ним контакт на транспортном и сеансовом уровнях эталонной модели взаимодействия открытых систем OSI/RM (в терминах TCP/IP (transmission control protocol / internet protocol) это – TCP-уровень). Взаимодействие на более низких уровнях (канальном, сетевом) происходит прозрачно для обеих

сторон. Протокол SMTP – протокол прикладного уровня и базируется поверх TCP. В его рамках не оговариваются ни размер сегментов данных, ни правила квитирования, ни отслеживание ошибок, возникающих при передаче информации.

Итак, по уже установленному соединению клиентское ПО передает команды SMTP-серверу, ожидая тут же получить ответы. В арсенал SMTP-клиента, равно как и сервера, входит около 10 команд, но воспользовавшись только пятью из них, уже можно легально послать почтовое сообщение, это HELO, MAIL, RCPT, DATA, QUIT. Их использование подразумевается именно в такой последовательности. HELO (усеченная форма от hello, приветствие) предназначена для идентификации отправителя, MAIL указывает адрес отправителя, RCPT (от англ. recipient – принимающий) – адрес назначения. После команды DATA и ответа на нее клиент посылает серверу тело сообщения, которое должно заканчиваться строкой, содержащей лишь одну точку. Выражаясь языком программистов, сервер узнает о прекращении посылки сообщения, встретив следующий набор символов "\r\n.\r\n" или "<CRLF>.<CRLF>" (CRLF – так называемый возврат каретки с переходом на новую строчку, обычно такое получается при нажатии клавиши Enter во многих редакторах).

Для демонстрации вышесказанного можно воспользоваться программой **telnet** (**ssh**). Если выполнить в командной строке Unix-подобных операционных систем следующую команду:
telnet name_of_mail_server.ru 25
или, нажав Пуск / Выполнить, ввести
telnet name_of_mail_server.ru 25
(для пользователей Windows), то установится TCP-соединение с 25-м портом указанного сервера (естественно, вместо слов name_of_mail_server следует набрать имя почтового сервера). Этот TCP-порт служит для обеспечения взаимодействия по протоколу SMTP (например, 80-й TCP-порт обычно закреплен за веб-сервером, принимающим и отсылающим информацию по HTTP-протоколу). Если не будет видно набираемых символов в сеансе **telnet**, попробуйте изменить параметры вывода на экран (Параметры / Отображение ввода).
В следующем примере приведен telnet-сеанс общения с несуществующим public_mail_server.ru, абонентом которого является некто с именем test_only, ему и предназначается почтовое сообщение.

```
220 ESMTP PUBLIC MAIL SERVER.RU Thu, 25 Apr 2002 14:15:50 +0400
helo my_comp.ru
250 public_mail_server.ru Hello as1-52.dialup.onego.ru
[195.161.137.53]
mail from: me@my comp.ru
250 <me@my comp.ru> is syntactically correct
rcpt to: test only@public mail server.ru
250 <test_only@public_mail_server.ru> verified
data
354 Enter message, ending with "." on a line by itself
body of the test message from me@my comp.ru to
test only@public mail server.ru
.
250 OK id=170gLG-000FEI-00
quit
```

Непосредственно после установления соединения сервер выдает строчку с кодом ответа 220. В ответ на нее клиент может инициировать сеанс связи по протоколу SMTP, послав команду HELO (можно маленькими буквами) и указав у нее в аргументах имя своего компьютера. Сюда можно писать все, что вздумается, потому что по принятии команды HELO сервер обязан сделать запрос в DNS и, если это возможно, по IP-адресу определить доменное имя компьютера клиента. (IP-адрес уже известен на момент установления соединения по протоколу TCP.) Именно поэтому в строчке "250 public_…" присутствует имя as1-52.dialup.onego.ru, соответствующее одному из IP-адресов модемного пула провайдера, а не my_comp.ru, как было указано на стадии приветствия.

Далее в команде "MAIL FROM:" клиент сообщает обратный адрес отправителя, который проверяется обычно только на корректность (это зависит от настроек SMTP-сервера). После слов "RCPT TO:" следует набрать адрес электронной почты абонента на данном сервере. Ответ "250 <test_only@public_mail_server.ru> verified" свидетельствует о существовании логина с именем test_only. Клиент отсылает команду DATA и ждет приглашения начать пересылку тела письма (код 354).

Сообщение может быть достаточно длинным, но обязательно должно заканчиваться строкой, в которой есть одна-единственная точка. Это служит сигналом SMTP-серверу о том, что тело письма закончилось. Он присваивает этому письму определенный идентификатор и ждет команды QUIT, после чего сеанс считается завершенным.

Как будут действовать пользовательское и серверное ПО в случае необходимости "массовой" рассылки? Если клиент посылает сообщение, у которого в заголовочной части в поле CC указаны несколько e-mail адресов, первый по пути следования SMTP-сервер должен будет в общем случае установить сеанс продвижения почты с каждым из серверов данного списка и отослать точную копию письма каждому. В случае использования поля BCC клиент, формирующий сообщение, уничтожит запись BCC в теле сообщения и несколько раз (по количеству адресатов) отошлет первому SMTP-серверу команду "RCPT TO:" каждый раз с новым адресом в качестве аргумента. Таким образом, сервер получит указание разослать почту по многим адресатам. Причем в этом случае получатели писем ничего не будут знать друг о друге, т. к. рассылка осуществляется посредством команд SMTP-протокола (без использования информации в заголовочной части письма).

Что будет, если попробовать соединиться с каким-нибудь мейл-сервером и через него передать почту, предназначенную другому? (В предыдущем примере адресатом был test_only@public_mail_server.ru, поэтому устанавливалось соединение именно с public_mail_server.ru.) Чаще всего на ввод команды "RCPT TO:" в ответ вы получите "relaying denied" (пересылка запрещена). Сделано это с целью борьбы с неконтролируемыми и несанкционированными почтовыми рассылками.

В принципе, эта ситуация очень похожа на простое использование транзитных серверов исходящих сообщений. Вы указываете один компьютер в качестве следующего пункта пересылки, а сообщение предназначается другому. Поэтому при настройке почтовых серверов администраторами вводятся ограничения на круг IP-адресов и доменных имен, для которых работает пересылка. Если этого не сделать, их сервер сразу же попадет в черный список, используемый при фильтрации такими гигантами, как hotmail.com, yahoo.com, mail.com и почта от абонентов "проштрафившихся" адресов будет блокироваться.

Протокол SMTP существует достаточно давно – с 1982 года (RFC 821); впоследствии в него были внесены некоторые дополнения, выразившиеся в появлении ESMTP (Extended SMTP, RFC 1425; 1993 год). Если клиент поддерживает ESMTP, его первая команда будет EHLO (Extended Hello), а не HELO. Сервер в ответ на нее должен выдать список всех дополнительных команд, которые он умеет обрабатывать. Клиент может ими воспользоваться, а может проигнорировать, послав сообщение продемонстрированным выше способом.

Контрольные вопросы к главе 3

1. Укажите положение в стеке **OSI/RM** следующих протоколов: **IP**, **TCP**, **UDP**, **ARP/RARP**, **ICMP**.
2. Что такое датаграмма?
3. На каких уровнях по модели **OSI/RM** для стека **TCP/IP** может применяться (обычно применяется) контроль качества переданной информации?
4. Как узнать длину поля "данные" в **TCP**-сегменте?
5. Что такое фрагментация **IP-**датаграмм? Какие механизмы существуют для поддержки фрагментации?
6. Объясните, почему широковещательный пакет сетевого уровня часто инкапсулируется в широковещательный кадр канального уровня. Подумайте, может ли возникнуть ситуация, когда его необходимо инкапсулировать в кадр канального уровня с уникальным (unicast) адресом получателя.
7. Опишите схему действий устройств, изображенных на рисунке в разделе 3.6, при отправке, перенаправлении и получении датаграмм.
8. Опишите схему действий устройств, изображенных на рисунке в разделе 3.6, при отправке, перенаправлении и получении датаграмм в случае, если добавлен еще один маршрутизатор между изображенным маршрутизатором и компьютером 2.
9. Опишите схему действий устройств, изображенных на рисунке в разделе 3.6, при установлении **TCP-**соединения между компьютерами 1 и 2 (в пояснениях следует затронуть транспортно-сеансовый, сетевой и канальный уровни).
10. Каким образом происходит подтверждение приема данных при общении по протоколу **TCP**?

Глава 4.
Средства и утилиты сетевого тестирования

4.1. ping

Программа ping предназначена для проверки доступности удаленного хоста или сетевой службы и часто используется в качестве самой первичной диагностики неисправностей (отсутствия связи) в глобальных и локальных сетях. Программа посылает **ICMP** (Internet Control Message Protocol) эхо-запрос на хост и ожидает возврата **ICMP** эхо-отклика. Время между посылкой **ICMP**-запроса и получением ответа может свидетельствовать о пропускной способности канала связи.

Программа ping реализована во всех сетевых операционных системах. С ее помощью можно тестировать сеть **IP-**пакетами любой длины от минимальной (64 байта) до максимально допустимой (~64 килобайтов) (длина пакета задается опциями в командной строке).

Следует также помнить, что во многих случая ping-пакеты могут блокироваться промежуточными маршрутизаторами, а хосты – пункты назначения могут быть сконфигурированы на запрет приема / отправки **ICMP-**сообщений.

Результат выполнения команды ping приведен ниже.

```
[alexmou@plasma alexmou]$ ping -c 3 localhost
PING localhost.localdomain (127.0.0.1) from 127.0.0.1 : 56(84) bytes of
data.
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=0 ttl=64
time=134 usec
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=1 ttl=64
time=155 usec
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=2 ttl=64
time=106 usec

--- localhost.localdomain ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max/mdev = 0.106/0.131/0.155/0.024 ms
```

В последней строке указывается время обращения **ICMP-**пакетов (минимальное / среднее / максимальное для нескольких пакетов и отклонение от среднего). Это время складывается из двойного

времени прохождения по сети и времени обработки пакета на удаленной стороне и промежуточных маршрутизаторах (usec обозначает микросекунды).

Определить пропускную способность сети можно, если построить график зависимости времени обращения от длины **ICMP-**сообщения. Построив точки, провести по ним прямую линию. Отсечка по вертикальной оси покажет время обращения пакета нулевой длины. Наклон будет свидетельствовать о пропускной способности канала до точки назначения.

Внимание! При посылке **ICMP-**пакета длиной более 1.5 кбайта в сетях Ethernet он будет дробиться на более мелкие фрагменты.

4.2. traceroute (tracert)

Программа traceroute позволяет посмотреть маршрут, по которому двигаются **IP**-датаграммы от одного хоста к другому. Обычно две последовательные датаграммы, отправленные от одного и того же источника к одному и тому же пункту назначения, проходят по одному и тому же маршруту, однако гарантировать это невозможно. С помощью traceroute можно также воспользоваться **IP-**опцией маршрутизации от источника.

traceroute, как и ping**,** использует **ICMP** и поле TTL в **IP-**заголовке. Поле TTL (время жизни) – это 8-битовое поле, в которое отправитель записывает конкретное значение (зависит от операционной системы).

Каждый маршрутизатор, который обрабатывает датаграмму, уменьшает значение TTL на единицу или на количество секунд, в течение которых маршрутизатор обрабатывал датаграмму. Так как большинство маршрутизаторов задерживает датаграмму меньше чем секунду, поле TTL, как правило, уменьшается на единицу и довольно точно соответствует количеству пересылок.

На хост назначения отправляется **IP**(**UDP**)-датаграмма 1 с TTL, установленным в единицу. Первый маршрутизатор, который должен обработать датаграмму 1, уничтожает ее (так как TTL равно 1) и отправляет **ICMP-**сообщение 2 об истечении времени (time exceeded). **ICMP-**сообщение инкапсулировано в **IP**-датаграмму 2, следовательно, отправитель изначальной **UDP**-датаграммы 1 всегда знает **IP**-адрес

отправителя 2. Таким образом определяется первый маршрутизатор по пути следования.

Затем traceroute отправляет датаграмму с TTL, равным 2, что позволяет получить IP-адрес второго маршрутизатора. Это продолжается до тех пор, пока датаграмма не достигнет хоста назначения. Однако если датаграмма прибыла именно на хост назначения, он не уничтожит ее и не сгенерирует **ICMP-**сообщение об истечении времени, так как датаграмма достигла своего конечного пункта назначения.

В **UDP-**датаграммах (User Datagram Protocol), которые посылает traceroute, устанавливается, например, неиспользуемый номер **UDP-**порта (больше чем 30000), что делает невозможным обработку этой датаграммы каким-либо приложением. Поэтому, когда прибывает подобная датаграмма, **UDP-**модуль хоста назначения генерирует **ICMP-**сообщение другого типа: "порт недоступен" (port unreachable).

Результат выполнения команды может содержать строки со звездочками. Они говорят о том, что пакет либо утерян (отброшен вследствие загруженности маршрутизаторов), либо маршрутизатор не имеет права отсылать **ICMP-**сообщение "time exceeded" (в случае * * *)

```
12 * bubble-gum.net (211.10.11.170) 187.258 ms 163.876 ms
13 * * *
14 www.star-systems.com (64.78.63.91) 223.622 ms * 260.829 ms
```

4.3. netstat

Программа netstat показывает текущее состояние различных структур данных, связанных с сетевым взаимодействием. В зависимости от выбранных опций можно узнать информацию о таблице маршрутизации (-r), текущих **TCP-**соединениях (-a), статистическую информацию попротокольно (-s).

Для сокетов TCP (просмотр с опцией -a) допустимы следующие состояния:

CLOSED	– закрытое, сокет не используется;
LISTEN	– ожидание входящих соединений;
SYN_SENT	– попытка установить соединение;

SYN RECEIVED	– процесс начальной синхронизации соединения;
ESTABLISHED	– соединение установлено;
CLOSE_WAIT	– удаленная сторона отключилась; ожидание закрытия сокета;
FIN_WAIT_1	– сокет закрыт; отключение соединения (отсылка флага FIN);
LAST_ACK	– удаленная сторона отключилась, сокет закрыт; ожидание подтверждения;
FIN_WAIT_2	– сокет закрыт; ожидание отключения от удаленной стороны (ожидание флага FIN);
TIME_WAIT	– ожидание после закрытия повторной передачи (два сообщения с флагом FIN) отключения удаленной стороны.

4.4. tcpdump

Утилита tcpdump используется для перехвата трафика, получаемого (отсылаемого) конкретным интерфейсом канального уровня (например, сетевой платой стандарта Ethernet) благодаря переводу его в специальный режим promiscuous.

В Unix-подобных операционных системах эта утилита находится в каталоге /usr/sbin/ и обычно запускается с правами суперпользователя.

Наиболее часто используемый вызов утилиты tcpdump выглядит следующим образом:

```
tcpdump [-c count] [-s snaplen] [-w file] [expression]
```

При этом в файл file будут записаны несколько (count) первых отловленных кадров, удовлетворяющих маске expression (например, not port 25). Причем если длина кадра была более snaplen, то он будет усечен до этой величины.

Начало созданного файла будет содержать заголовки библиотеки pcap (24 байта), а далее последовательно будут расположены кадры, каждому из которых предшествует временная метка (8 байт), полная длина кадра (4 байта) и длина захваченной части кадра (4 байта). Таким образом,

начало первого Ethernet кадра (**MAC-**адрес назначения) будет находиться по смещению 40 относительно начала файла. Формат заголовка всего файла и заголовка кадра можно посмотреть в заголовочном файле `pcap.h`.

Анализируя значение 13-го и 14-го байтов внутри каждого кадра, можно узнать, например, каков был протокол верхнего уровня по отношению к канальному (<1518 – длина кадра, 2048 (0800h) – **IPv4**, 0806h – **ARP**, 8035h – **RARP** и т. д., полный перечень зарегистрированных в IANA протоколов приведен по адресам:

http://standards.ieee.org/regauth/oui/index.shtml

или http://www.iana.org/assignments/ethernet-numbers).

В документах по указанным адресам можно также найти информацию о фирме производителя сетевого оборудования, считав три старшие байта **MAC**-адреса (например, используя команду ifconfig).

Примеры использования tcpdump:

а) Вывести все принятые и отправленные пакеты с **IP-**адресом 192.168.1.1:

```
tcpdump host 192.168.1.1
```

б) Вывести все **IP**-пакеты между хостом ace и всеми остальными, но не helios:

```
tcpdump ip host ace and not helios
```

в) Вывести первый и последний пакеты **TCP-**сеанса связи (SYN- и FIN-сегменты), в которых принимал участие компьютер с нелокальным **IP**-адресом:

```
tcpdump 'tcp[tcpflags] & (tcp-syn|tcp-fin) != 0 and not src and dst net localnet'
```

г) Вывести все **IP-**пакеты длиной более 576 байт, прошедшие через маршрутизатор snup:

```
tcpdump 'gateway snup and ip[2:2] > 576'
```

д) Вывести все пакеты с **IP-**адресами класса D (групповой рассылки), которые были инкапсулированы и посланы не в широковещательных кадрах и кадрах групповой рассылки Etehrnet:

```
tcpdump 'ether[0] & 1 = 0 and ip[16] >= 224'
```

е) Вывести все **ICMP-**пакеты, которые не были сформированы утилитой ping (не эхо-запросы и не эхо-ответы):

```
tcpdump 'icmp[icmptype] != icmp-echo and icmp[icmptype] != icmp-echoreply'
```

Контрольные вопросы к главе 4

1. Опишите, каким образом используется **ICMP-**протокол в сетевых утилитах ping и traceroute.
2. Устанавливается ли **TCP-**соединение между компьютером, на котором выполняется программа ping, и компьютером, которому посылается **ICMP** эхо-запрос? Почему?
3. Чем определяется сокет (соединение) в терминах **TCP/IP**? А в терминах **OSI/RM**?
4. Напишите выражение для фильтра утилиты tcpdump, который позволит отлавливать сетевые пакеты только с запросами к **DNS-**серверу (**UDP-**порт 53).

ЧАСТЬ II

РАЗРАБОТКА СЕТЕВЫХ ПРИЛОЖЕНИЙ В СРЕДЕ ОС UNIX

Глава 5.
Основные концепции организации ввода-вывода и управления процессами в ОС Unix

5.1. Подсистема ввода-вывода в ОС UNIX

Каждому устройству в Unix-подобных операционных системах ставится в соответствие так называемый "специальный файл". Все такие файлы, как правило, располагаются в каталоге `/dev`. Дав команду

`ls -l /dev` ,

вы можете посмотреть, какие именно файлы имеются в этом каталоге в вашей системе. Например, вы можете обнаружить там файлы `hda`, `hdb` – такие файлы соответствуют жестким дискам; разделам жестких дисков соответствуют файлы `hda1`, `hda2` и т. д. Такие названия для специальных файлов жестких дисков (с **IDE**-интерфейсом) и разделов на них приняты в ОС Linux. В других Unix-системах вы можете встретить иные названия. Общее количество специальных файлов довольно велико, поэтому здесь мы не будем описывать, какой файл какому устройству соответствует, отметим лишь, что иногда об этом можно догадаться по имени специального файла, например `/dev/mouse`, `/dev/audio` и т. п. Отметим еще, что в Unix выделяется

2 типа специальных файлов – файлы блочных устройств и файлы символьных устройств. На самом деле типов устройств 3 – блочные, символьные и сетевые интерфейсы, но последним из них не соответствуют никакие специальные файлы. Как мы увидим далее, это проявляется в некоторых особенностях при разработке сетевых приложений.

Подход, при котором каждому устройству ставится в соответствие некоторый файл, имеет положительное свойство, заключающееся в том, что для прикладных программ (точнее, для человека, их разрабатывающего) нет никакой разницы, откуда они читают данные или куда их записывают. Под словами "нет никакой разницы" имеется в виду, что для выполнения операций записи-чтения используются одни и те же системные вызовы (системный вызов – это обращение к операционной системе со стороны прикладной программы) вне зависимости от того, происходит обмен данными с обычным дисковым файлом или с каким бы то ни было устройством.

Для осуществления обмена данными с устройствами в прикладной программе используются следующие системные вызовы. Для открытия устройства (или файла) используется системный вызов `open()`, возвращающий так называемый дескриптор файла. Этот дескриптор представляет собой целое неотрицательное число, являющееся индексом в таблице файлов, открытых процессом. В дальнейшем этот дескриптор используется для любой операции с файлом или устройством. Следует заметить, что сетевое взаимодействие, видимо, не совсем вписалось в парадигму "Everything In Unix Is A File", поэтому для открытия "файла", представляющего собой какой-то ресурс на удаленном узле, используется другая функция, а именно `socket()` в сочетании с некоторыми другими. Однако возвращаемый этой функцией дескриптор вполне такой же, он используется как индекс в той же таблице, поэтому сама передача данных для сокетов (более подробно этот термин поясняется ниже) практически не отличается от передачи данных при работе как с обычными файлами, так и с различными устройствами через специальные файлы. После окончания работы с файлом (устройством, сокетом) его полагается закрывать, что делается с помощью вызова `close()`. Чтение данных производится посредством вызова `read()`, запись – посредством `write()`. Отметим, что существуют операции, не являющиеся операциям чтения или записи собственно данных (например, установка скорости передачи для последовательного порта и т. п.). Для выполнения таких операций предназначены вызовы `ioctl()` и `fcntl()`.

Следует отметить, что в стандартной библиотеке для языка программирования Си имеется большое количество функций, являющихся "надстройками" над системными вызовами, перечисленными в предыдущем абзаце. В качестве примеров таких надстроек можно привести группу `fopen()/fclose()/fread()/fwrite()`, реализующую ввод-вывод с буферизацией (часто говорят потоковый ввод-вывод), функции `scanf()` и `printf()`, выполняющие, соответственно, форматированный ввод из стандартного устройства ввода и вывод в стандартное устройство вывода, функции `fscanf()` и `fprintf()`, выполняющие, соответственно, форматированный ввод из произвольного потока ввода и форматированный вывод в произвольный поток вывода, а также другие группы функций.

При дальнейшем изложении будет применяться следующая терминология. Число, возвращаемое функцией `open()`, будет называться "дескриптор файла" или "дескриптор устройства". Число, возвращаемое функцией `socket()`, будет называться "дескриптор сокета". Слова "дескриптор ввода-вывода" или "дескриптор потока ввода-вывода" будут означать любой из вышеприведенных терминов: или "дескриптор файла", или "дескриптор устройства", или "дескриптор сокета". Вместо "дескриптор ввода-вывода" часто будет употребляться просто "дескриптор", без "ввода-вывода". Слово "сокет" будет употребляться в смысле "совокупность данных, описывающих (внутри и для целей ОС) параметры и состояние конечной точки соединения двух программ". Такая терминология обусловлена тем, что в любом процессе (то есть в исполняющейся программе) обычные дисковые файлы, устройства и сокеты представлены одинаково, а именно как индексы в таблице файлов, открытых этим процессом.

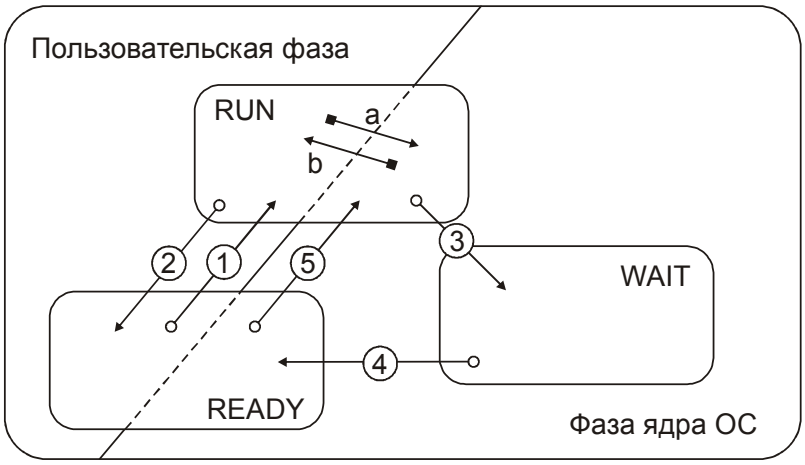
Заметим еще, что дескрипторы уникальны в рамках только одного процесса. Это означает, что если у одного процесса дескриптор со значением, скажем, 7 представляет какой-то дисковый файл, то совершенно не обязательно, что в другом процессе он представляет этот же файл – дескриптор с этим же значением может представлять другой файл (или устройство, или сокет). Однако следует обратить внимание, что при операции создания процесса (см. раздел 8.1) таблица открытых файлов (по умолчанию) наследуется дочерним процессом, поэтому, по крайней мере сразу же после создания процесса, все дескрипторы дочернего процесса будут представлять в точности те же сущности (файлы, устройства, сокеты), что и у родительского процесса.

В заключение этого раздела отметим, что сразу после запуска процесс имеет три открытых потока ввода-вывода. Первый из них – устройство ввода по умолчанию ("стандартное" устройство ввода), ему соответствует дескриптор 0; второе – устройство вывода по умолчанию ("стандартное" устройство вывода), ему соответствует дескриптор 1; третье – устройство вывода сообщений об ошибках, ему соответствует дескриптор 2.

5.2. Подсистема управления процессами

Важным понятием в теории операционных систем и программирования является понятие процесса (или задачи). Будем понимать под задачей процесс исполнения некоторой программы. Понятно, что в многозадачных средах, каковыми и являются многие операционные системы, в том числе и Unix-подобные, задачи должны иметь возможность исполнятся одновременно (при наличии нескольких процессоров) или "псевдоодновременно" (если процессор один) в режиме разделения времени. Одновременно в буквальном понимании этого слова могут исполняться столько процессов, сколько имеется процессоров в вычислительной системе. Другие процессы, которых может быть очень много, в это время не исполняются, а ожидают своего времени. В связи с этим говорят о состояниях процессов (задач). Вообще, планирование задач – это большая и сложная тема, которая в этом пособии по понятным причинам рассматриваться не будет. Для понимания материала, излагаемого в дальнейшем, нам необходимо будет рассмотреть лишь вопрос о том, в каких именно состояниях могут находиться процессы и при каких обстоятельствах они совершают переходы из состояния в состояние. Подробности, касающиеся алгоритмов планирования задач, механизмов их переключения, как не относящиеся к тематике данного пособия, в него не включены.

Ниже на рисунке приведена диаграмма состояний процессов в многозадачных средах. Как видно, всего имеется 3 состояния – состояние исполнения (RUN), состояние ожидания (WAIT) и состояние готовности (READY). Состояние ожидания часто называют состоянием сна (SLEEP).



Состояние исполнения – это такое состояние, в котором тот или иной процесс занимает процессор, то есть исполняется в данный момент времени. Как уже отмечалось, максимальное число процессов в таком состоянии равно числу процессоров в системе.

Состояние готовности – это такое состояние, в котором процессу для того, чтобы продолжить работу, необходимо только процессорное время. В таком состоянии может находиться много процессов; все они в совокупности образуют очередь ожидания процессорного времени (очередь процессов, готовых к исполнению).

Состояние ожидания (сна) – это состояние, в котором для того, чтобы продолжить работу, процессу необходимо не только процессорное время, но еще и другие ресурсы или данные. Например, процесс может ожидать ввода данных с клавиатуры или из сети, ожидать освобождения какого-либо ресурса, используемого в этот момент другим процессом, ожидать загрузки страницы памяти, оказавшейся выгруженной на внешний носитель в результате действий системы подкачки страниц и т. п. Процессы, ожидающие наступления того или иного события, также образуют очереди.

Рассмотрим теперь переходы между состояниями. Переход "1" – переход из состояния готовности в состояние исполнения; происходит тогда, когда процесс, готовый к продолжению работы, получил процессорное время. Переход "2" – обратный переход, так называемое

вытеснение; происходит тогда, когда процесс исчерпал очередной квант времени, отведенный ему, и не перешел при этом в состояние ожидания. Переход "3" – переход из состояния исполнения в состояние ожидания (сна); процесс переходит в это состояние, когда операция, которую он хотел исполнить, не может быть выполнена немедленно. Переход "4" – переход из состояния ожидания в состояние готовности; операция, которая была отложена при переходе "3", теперь может быть выполнена, нужно только дождаться своей очереди на процессорное время.

Буквами "a" и "b" помечены переходы из так называемой пользовательской фазы (user space) в так называемую фазу ядра (kernel space) и обратно; эти переходы не являются переходами из состояния в состояние. Переход "a" – это и есть, собственно, системный вызов, обращение процесса к операционной системе; переход "b" – возврат из системного вызова. Название "пространство" ("space") связано с тем, что, исполняясь в этих двух фазах, процесс имеет разные стеки, разные права доступа к различным участкам памяти, разные привилегии по отношению к возможности обращения к аппаратным средствам.

5.3. Взаимосвязь подсистем ввода-вывода и управления процессами

Рассмотрим теперь, какое отношение имеет ввод-вывод к управлению процессами. Прежде всего, сразу отметим, что переходы из состояния исполнения в состояние ожидания происходят чаще всего именно в связи с тем, что та или иная операция ввода-вывода не может быть завершена в момент попытки ее выполнить. Например, операция чтения не может быть выполнена по тривиальной причине отсутствия данных для чтения. Операция записи может быть отложена в связи с тем, что некоторые внутренние буферы операционной системы (драйверов устройств) или буферы в аппаратуре могут быть в момент попытки записи данных полностью заполненными.

Переход в состояние ожидания при невозможности немедленно завершить какую-либо операцию ввода-вывода – это вполне разумное (в смысле экономии процессорного времени) решение; по умолчанию процессы так ведут себя всегда.

Представим, однако, такую ситуацию, когда процессу нужно одновременно работать с несколькими дескрипторами попеременно. Такие

ситуации – обычное дело; например, представьте, что вы разрабатываете программу-клиент для службы типа ICQ (популярная система "мгновенной" доставки сообщений). Такая и ей подобные программы должны будут работать, по крайне мере, с двумя дескрипторами (не считая устройства стандартного вывода) – дескриптором устройства ввода по умолчанию (в данном случае с клавиатурой) и с дескриптором сокета (общение с сервером осуществляется посредством сокета).

Предположим, процесс обратился к операционной системе на предмет чтения данных из сети и при этом оказалось так, что в этот момент данных нет. Далее пользователь программы захотел что-либо ввести с клавиатуры. Процесс, однако, не может принять имеющиеся данные, поскольку он находится в состоянии сна внутри системного вызова, который должен прочитать данные из сети и ждет, пока они не придут. В результате пользователь будет иметь возможность отправить данные только после того, как из сети придут данные, предназначенные ему. Понятно, что такой программой будет пользоваться крайне неудобно.

Еще один типичный пример, когда процессу необходимо каким-то образом мультиплексировать ввод-вывод (то есть получать и отправлять данные более чем через один дескриптор) – это всевозможные программы-серверы; им, как правило, приходится обслуживать несколько клиентов одновременно, а каждому клиенту соответствует свой сокет, представленный в сервере отдельным дескриптором ввода-вывода.

Здесь нужно отметить, что существует такое понятие, как "имя сокета", которое не следует путать с понятием "сокет" или с понятием "дескриптор сокета". Имя сокета – это совокупность двух чисел, **IP**-адреса и номера **TCP-** или **UDP**-порта. Кроме того, в сокет входит имя сокета с другой стороны соединения. Одна программа может одновременно иметь несколько сокетов с одинаковыми именами с локальной стороны; каждому из этих сокетов будет соответствовать собственный дескриптор.

Например, любой сервер будет иметь по крайней мере 2 сокета, один из которых находится в режиме прослушивания и служит для принятия соединений от клиентов, а другой используется непосредственно для обмена данными с ними. Локальное имя у этих сокетов будет одинаковое, а дескрипторы им будут соответствовать разные. Кроме того, в случае, если сервер может обслуживать параллельно несколько клиентов, то сокетов для обмена данными может существовать более одного. Все они будут иметь различные **IP**-адреса и **TCP**-порты удаленной

стороны; каждому такому сокету будет соответствовать свой файловый дескриптор.

В следующей главе будут рассмотрены различные способы организации программы в тех случаях, когда ей необходимо обмениваться данными через несколько дескрипторов ввода-вывода одновременно.

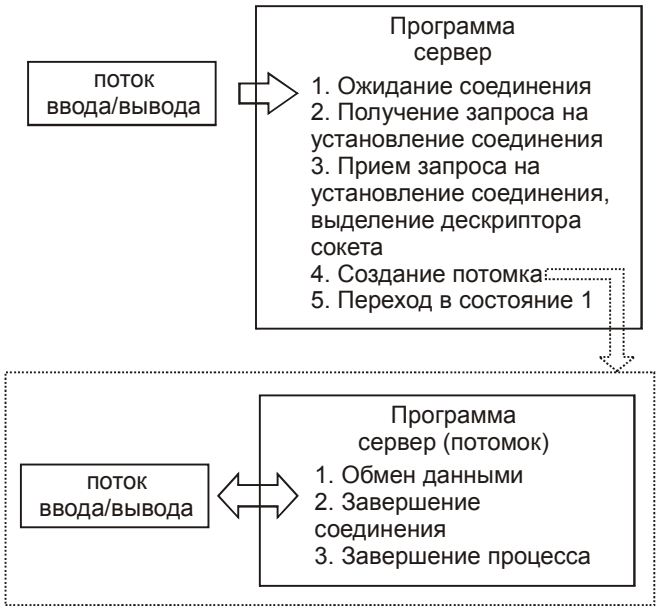
Контрольные вопросы к главе 5

1. Перечислите системные вызовы ОС Unix, предназначенные для обмена данными с устройствами и управления ими.
2. Сколько открытых потоков ввода-вывода имеет процесс сразу после своего запуска? Какие это потоки?
3. Как вы понимаете выражение "everything in Unix is a file" (все в Unix – это файл)?
4. Что такое дескриптор сокета (устройства)?
5. Нарисуйте диаграмму состояний процесса в многозадачной среде. Какие возможны переходы между состояниями и как они называются?
6. Поясните выражения "процесс исполняется в пользовательской фазе", "процесс исполняется в системной фазе". Чем различаются эти фазы исполнения процесса?
7. Приведите пример программы, которая не переходит в системную фазу.
8. Почему процессы переходят в состояние ожидания (сна)?
9. Продемонстрируйте на примере какой-либо программы необходимость мультиплексирования потоков ввода-вывода.

Глава 6.
Стратегии организации ввода-вывода в среде ОС UNIX

6.1. Многопроцессный подход

Как было описано в конце предыдущей главы, в случае, если одному процессу необходимо обмениваться данными с несколькими устройствами или сокетами одновременно, одними только функциями непосредственно ввода-вывода не обойтись. Ничто, однако, не мешает использовать для обмена данными с каждым устройством отдельный процесс или нить (нить, или легковесный процесс, – это поток управления внутри процесса, своего рода подпроцесс). Иными словами, вместо того, чтобы организовывать мультиплексирование ввода-вывода в одном процессе, можно возложить работу по обмену данными с каждым устройством (или сокетом) на отдельный процесс или нить, то есть сделать программную систему многопроцессной или многонитевой.



В качестве примера рассмотрим в общих чертах программу-сервер, организованную таким способом. Число сокетов у сервера равно числу клиентов, обслуживаемых в данный момент времени плюс еще один, необходимый для принятия соединений от новых клиентов. Алгоритм такого многопроцессного или многонитевого (многопоточного) сервера будет выглядеть примерно следующим образом. После проведения подготовительной работы сервер ожидает соединения от клиентов с помощью вызова `accept()` (подробности см. в главе 8). Как правило, сразу после установления соединения сервер порождает дочерний процесс (или дополнительную нить) и "поручает" ему обслуживание данного клиента (напомним, что при создании процесса все дескрипторы ввода-вывода наследуются дочерним процессом; при использовании нитей все глобальные переменные доступны всем нитям). Таким образом, каждый экземпляр сервера (или каждая его нить) обменивается данными только с одним клиентом, то есть работает с одним сокетом (дескриптором), поэтому необходимость в мультиплексировании отпадает. В случае использования нескольких процессов "главный" из них после создания дочернего, как правило, закрывает сокет, предназначенный для обмена данными с клиентом, а дочерние процессы закрывают сокет, предназначенный для принятия соединений (помним, что дескрипторы наследуются).

Указанный подход обладает, однако, рядом недостатков. Во-первых, он несколько расточителен (при большом числе одновременно обслуживаемых клиентов) в отношении использования процессорного времени и памяти, поскольку каждый экземпляр сервера занимает в оперативной памяти некоторое место и, кроме того, операция создания нового процесса может занимать значительное по масштабам процессора время. Во-вторых, иногда экземплярам сервера вне зависимости от того, реализованы они в виде отдельных самостоятельных процессов или в виде нитей в одном процессе, необходимо обмениваться с той или иной целью какими-то данными или, в общем случае, использовать какие-то ресурсы, которые нужны всем экземплярам сервера. Как известно, участок кода программы, в котором она тем или иным образом использует некоторый ресурс, который может использоваться другим процессом, исполняющим эту же или, возможно, другую программу, называется критической секцией. Такие участки кода нужно защищать соответствующими средствами синхронизации, такими как, например, семафоры или мьютексы. Это необходимо делать во избежание ситуации, когда в критической секции оказывается более одного

процесса (или нити), что может привести как минимум к получению неправильных результатов, а в некоторых случаях и к более серьезным последствиям, например к нарушению целостности совместно используемого ресурса с дальнейшим плохо предсказуемым результатом. Следует упомянуть, что сама задача выявления критических секций в программе может быть иногда чрезвычайно сложной.

К преимуществам многопроцессного (многонитевого) подхода можно отнести следующее:

- эффективность при работе на многопроцессорных машинах; каждый из процессов (нитей) может исполняться на своем процессоре

- "справедливое" обслуживание клиентов в случае, если выполнение запроса требует длительного времени; если один из процессов (нитей) занят формированием ответа на очередной запрос клиента, то он не будет монопольно занимать процессор, поэтому другие клиенты тоже получат свою долю времени; в качестве примера можно привести выдачу файла большого размера.

6.2. Мультиплексирование ввода-вывода в одном процессе

Альтернативой подходу, описанному в предыдущем разделе, является такой способ организации ввода-вывода, при котором с потоками данных, поступающих от нескольких устройств или сокетов, обменивался бы единственный экземпляр программы, один процесс. Таких способов существует, вообще говоря, несколько. Первый способ состоит в том, чтобы организовать мультиплексрование посредством поочередного опроса дескрипторов ввода-вывода на предмет готовности к выполнению той или иной операции. Второй способ состоит в использовании средств операционной системы, специально предназначенных для мультиплексирования ввода-вывода. Третий способ состоит в том, чтобы опрос не производить, но организовать программу так, чтобы она получала от операционной системы некоторое уведомление в случае, если те или иные дескрипторы станут готовы к выполне-

нию той или иной операции ввода-вывода. Это осуществляется при помощи механизма так называемых сигналов.

Общими недостатками однопроцессного подхода являются:

- неэффективное использование процессорного времени при работе на многопроцессорных машинах; процесс работает только на одном процессоре
- "несправедливое" обслуживание клиентов в случае, если отработка запроса занимает длительное время

Зачастую серверы пишут, применяя различные комбинации многопроцессного (многонитевого) подхода и подхода с использованием мультиплексирования в одном процессе. Такие комбинированные схемы подразумевают, что обслуживанием клиентов занимаются несколько процессов или нитей, при этом каждый процесс или нить обслуживают, в свою очередь, несколько клиентов.

В последующих подразделах описаны все 3 стратегии организации ввода-вывода.

6.2.1. Использование неблокирующего ввода-вывода

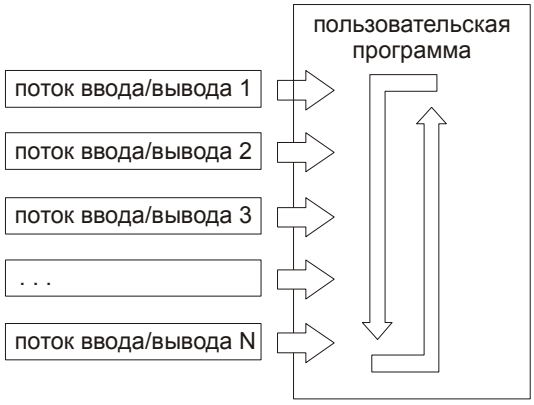
Как говорилось выше, в ситуациях, когда какая-либо операция не может быть выполнена немедленно, то есть сразу же после запроса на ее выполнение, процесс переходит в состояние ожидания. Заметим, что это происходит, как правило, внутри кода драйверов и других компонентов операционной системы и вполне прозрачно для прикладного программиста.

Однако такое поведение не является единственно возможным. Вторая возможность состоит в том, чтобы не переходить в состояние ожидания, а остаться в состоянии исполнения, но вернуться из фазы ядра в пользовательскую фазу, то есть завершить системный вызов. При этом, естественно, нужно каким-то образом уведомить процесс о том, что операция не может быть выполнена немедленно. Делается это стандартным для системных вызовов образом: вызов возвращает константу `-1`, а в глобальную переменную `errno` помещается код ошибки. Для рассматриваемого случая значение этой переменной будет равно константе `EAGAIN` (или `EWOULDBLOCK`).

Итак, если операция не может быть выполнена в момент обращения к операционной системе, процесс может либо перейти в состояние SLEEP, либо остаться в состоянии RUN, но вернуться в пользовательскую фазу (см. рис. в предыдущей главе).

Если речь идет об операциях ввода-вывода, то первый вариант поведения называется "блокирующий ввод-вывод" (blocking i/o), а второй, соответственно, – "неблокирующий ввод-вывод" (non-blocking i/o). Отметим еще раз, что по умолчанию ввод-вывод – блокирующий, поскольку такое поведение более разумно. Можно, однако, перевести то или иное устройство (или сокет) в режим неблокирующего ввода-вывода. Это делается с помощью вызова `fcntl()` с соответствующими аргументами. Переведя все каналы ввода-вывода в такой режим, можно организовать работу с ними в одном процессе (или нити) следующим образом: обращаться в цикле ко всем дескрипторам; если тот или иной дескриптор готов, операция выполнится, если нет, то вызов тут же завершится, и мы перейдем к следующему дескриптору.

Этот алгоритм схематически изображен на нижеприведенном рисунке.



Сразу бросается в глаза самый главный недостаток такого способа: процесс большую часть времени будет производить "холостые" обращения к устройствам (или сокетам), что является весьма расточительным в отношении использования процессорного времени и, таким образом, негативно сказывается на общей производительности системы. Поэтому использовать неблокирующий ввод-вывод настоятельно не

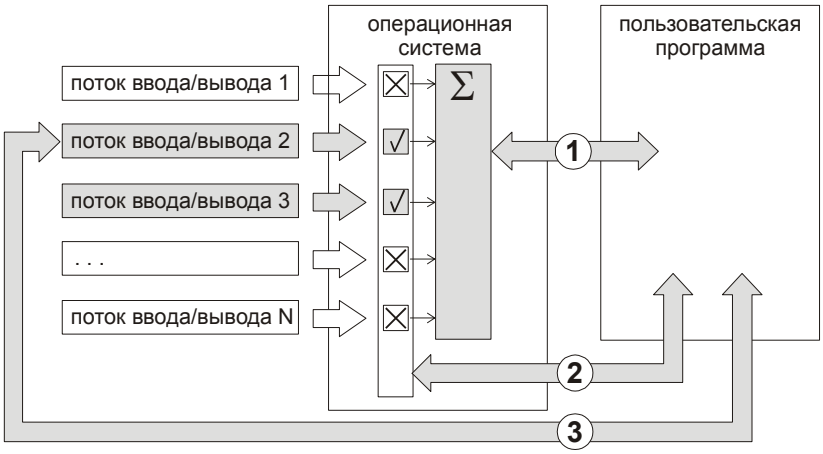
рекомендуется, за исключением случаев, где по-другому сделать невозможно.

6.2.2. Мультиплексирование ввода-вывода с использованием select() или poll()

Проблема в использовании неблокирующего режима для организации параллельной работы с несколькими дескрипторами состоит, очевидно, в том, что процессу необходимо обращаться ко всем дескрипторам по очереди; за один системный вызов проверяется состояние только одного дескриптора.

Существуют, однако, системные вызовы, которые позволяют прикладной программе опросить за один раз большое количество дескрипторов. Такими вызовами являются `select()` и `poll()`. Они принимают в качестве аргументов наборы дескрипторов и возвращаются тогда, когда хотя бы один из дескрипторов становится готовым к исполнению указанной операции ввода-вывода. Примеры использования этих вызовов будут приведены в дальнейшем.

На рисунке ниже схематически отображена общая идея программы, эксплуатирующая способ мультиплексирования ввода-вывода с использованием средств, предоставляемых операционной системой.

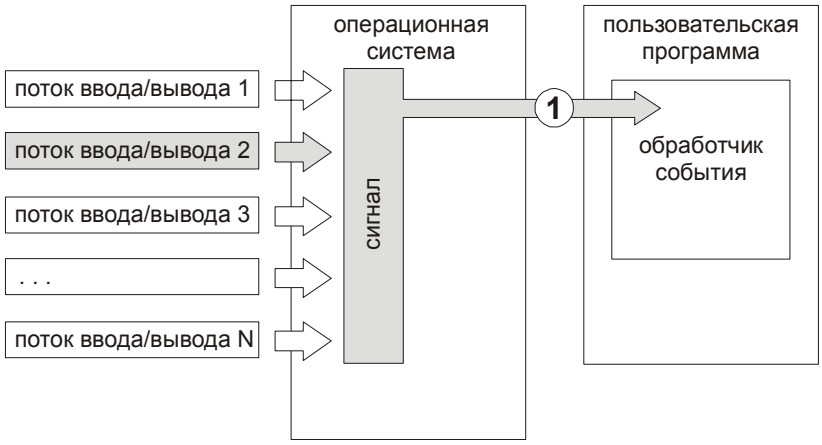


При использовании этих средств процесс при помощи вызовов `select()` или `poll()` определяет, сколько дескрипторов готово к той или иной операции ввода-вывода (на схеме это символически обозначено стрелкой с цифрой 1), затем определяет, какие именно дескрипторы готовы (стрелка с цифрой 2) и выполняет соответствующие операции над готовыми дескрипторами (стрелка с цифрой 3).

6.2.3. Использование механизма сигналов по стандарту POSIX

Обычно программу пишут таким образом, что если в ней необходимо выполнить какое-либо действие (имеются в виду не вычисления), то она делает обращение к операционной системе. Наряду с таким способом организации взаимодействия прикладной программы с ядром операционной системы существует и другой. Он заключается в том, что программа содержит одну или несколько подпрограмм, которые являются обработчиками некоторых событий. От программы требуется сообщить операционной системе адреса таких подпрограмм-обработчиков, после чего при возникновении указанных событий система передаст управление этим подпрограммам.

На рисунке ниже приведена упрощенная схема обмена данными между прикладной программой и внешними устройствами при использовании механизма обработчиков событий.



Суть данной схемы состоит в том, что при возникновении какого-либо события операционная система вызывает обработчик этого события (закрепленный за ним в операционной системе или переопределенный внутри пользовательской программы).

В терминологии операционных систем семейства Unix для обозначения событий и подпрограмм для их обработки употребляются понятия "сигнал" и, соответственно, "обработчик сигнала". В терминологии операционных систем семейства Windows для обозначения событий и подпрограмм для их обработки употребляются понятия "сообщение" и, соответственно, "процедура окна". В связи с этим часто говорят "послать сигнал (или сообщение)". Следует отметить, что, в отличие от действительно передачи сообщений (пакета байтов из одного адресного пространства одного процесса в адресное пространство другого), посылка сигнала есть передача управления его обработчику, а данные ему передаются обычным образом как параметры подпрограммы. Например, когда вы нажимаете на клавиатуре комбинацию control-c, то вызывается обработчик сигнала `SIGINT` (прервать работу) соответствующего процесса, который может выполнить какие-то действия; по умолчанию обработчик этого сигнала завершает процесс.

Перед рассмотрением вопроса о том, как с помощью механизма сигналов организовать ввод-вывод, приведем некоторые общие сведения, касающиеся обработки сигналов вообще. Как уже было сказано, обработчик сигнала – это обычная функция. Отличие от функций, которые не являются обработчиками сигналов, состоит в том, что

последние вызываются в программе явно, а обработчики сигнала – неявно, то есть не самой программой, а операционной системой. Очень важное следствие этого – обработчики сигнала вызываются асинхронно. Это означает, что в большинстве случаев нельзя предсказать, в каком именно месте программы придет сигнал и будет вызван обработчик. Из этого, в свою очередь, следует, что не всякое действие допустимо внутри обработчика, поэтому при написании его кода нужно соблюдать ряд предосторожностей; некоторые из них описаны ниже.

Если к тому моменту, когда ядро операционной системы смогло передать управление обработчику сигнала, возникло несколько сигналов одного типа, то обработчик будет вызван только один раз. Об этой особенности говорят как о "слиянии сигналов". Кроме того, процесс может сам заблокировать сигнал; если в течение промежутка времени, пока сигнал данного типа был заблокирован, возникнет несколько таких сигналов, то после разблокировки обработчик также будет вызван только один раз.

Это нужно иметь ввиду при написании обработчиков. В частности, если родительскому процессу интересен код возврата дочернего, то он должен ожидать его завершения. Если при этом ожидание производится по сигналу (SIGCHLD), то факт вызова обработчика обозначенного сигнала говорит лишь о том, что завершился по крайней мере один процесс; соответственно, алгоритм обработчика должен это учитывать. Подробнее об ожидании дочерних процессов см. в разделе 7.1.

Стандарт POSIX, однако, содержит несколько иной механизм сигналов, так называемый "сигналы реального времени". В отличие от обычных сигналов, сигналы реального времени не сливаются, а ставятся в очередь. Подробнее об этом рассказано в следующем разделе.

Если в обработчике вызывается какая-либо функция, которая вызывается и в основной программе, то она должна быть реентерабельной. Под этим понимается следующее: пусть выполнение функции в основной программе прервалось, а обработчик тоже ее вызывает. Если при этом не может возникнуть никаких нежелательных побочных эффектов, то такая функция реентерабельна (в нее можно входить более одного раза). В противном случае функция является нереентерабельной, то есть в нее нельзя войти до тех пор, пока она не завершилась. Нереентерабельной может быть любая функция, которая помимо стековой памяти использует любую другую. В качестве примеров нереентерабельных функций

из стандартной библиотеки для языка программирования Си можно привести функцию получения информации об узле по его доменному имени `gethostbyname()`, которая будет описана в разделе 9.6, функцию форматированного вывода в файл `fprintf()`; во многих системах нереентерабельны функции для выделения и освобождения памяти (`malloc()` и `free()`).

Безусловно, приведенные сведения о сигналах и их обработке ни в коей мере не являются полными. Авторы, однако, не ставили своей целью создать полное руководство по программированию в среде операционных систем семейства Unix; все сказанное выше о сигналах является лишь некоторым ознакомительным материалом и своего рода стимулом к более детальному изучению рассматриваемых вопросов. В качестве полного руководства можно порекомендовать информацию по стандартной библиотеке языка программирования Си (нужно набрать в командной строке info libc), а лучшим справочником по конкретным функциям является, по мнению авторов пособия, стандартная справочная система Unix (команда man). Как правило, справочные страницы по системным вызовам находятся во втором разделе (например, man 2 open), а страницы по функциям из стандартной (и прочих) библиотек – в третьем разделе.

Строго говоря, то, о чем пойдет речь дальше, не является мультиплексированием в том смысле, в котором это понимается при использовании `select()` или `poll()`, описанных в предыдущем разделе; однако механизмы сигналов позволяют организовать работу с несколькими дескрипторами одновременно, то есть решить те же задачи, что и при использовании системных вызовов, осуществляющих собственно мультиплексирование.

Отметим, что помимо механизма сигналов по стандарту POSIX имеется еще и стандартный механизм сигналов Unix (стандарт ANSI C). Нас будет интересовать только первый по той причине, что лишь при его использовании возможна передача файлового дескриптора обработчику сигнала. При использовании же механизма сигналов по стандарту ANSI C обработчику передается только номер сигнала (на случай, если один обработчик установлен на несколько сигналов сразу), поэтому при использовании этого механизма обработчик должен определить, какой именно из дескрипторов готов к операции ввода-вывода, например используя вызовы `select()` или `poll()`.

Для того чтобы осуществить ввод-вывод по сигналу, необходимо в программе проделать следующее. Во-первых, она должна содержать функцию, которая будет использована в качестве обработчика сигнала `SIGIO` (`SIGPOLL`). Именно этот сигнал получает процесс, когда появляется возможность выполнить ввод или вывод данных без ожидания. Для установки обработчика следует использовать вызов `sigaction()`. Стандарт ANSI C определяет для этих целей функцию `signal()`, но, как уже было отмечено, в этом случае нет возможности передать обработчику сигнала никакой информации, кроме идентификатора сигнала. Далее следует перевести нужные дескрипторы в режим асинхронного ввода-вывода. После этого обработчик сигнала `SIGIO` будет получать управление всякий раз, когда обозначенные дескрипторы окажутся готовыми к чтению или записи. Имеется еще ряд тонкостей, но они будут рассмотрены в следующей главе, в которой будут приведены конкретные примеры программ.

6.2.4. Прочие способы

Рассмотренные в предыдущих трех подразделах программные механизмы, позволяющие прикладным программам получать уведомления о возможности совершения операций ввода-вывода и организовать на их основе работу со многими дескрипторами ввода-вывода, являются традиционными, устоявшимися и реализованы практически в любой разновидности **Unix**-подобных операционных систем.

Наряду с этим в различных модификациях ядер таких систем имеются (и появляются) свои механизмы, предназначенные для достижения аналогичных целей. Понятно, что их использование может сделать программу непереносимой между различными ОС **Unix**, но при этом можно получить, например, такое преимущество, как высокая производительность, что имеет важное значение для сильно загруженных серверов. Далее будут рассмотрены несколько механизмов уведомления, о которых полезно иметь представление при разработке приложений в среде ОС **Unix** [23].

В ОС **Solaris** имеется механизм, доступный для приложений через псевдоустройство /dev/poll. В общих чертах его использование таково: прикладная программа открывает обозначенное псевдоустройство, записывает в него дескрипторы, состояние которых нужно отслеживать, и затем при чтении из него "драйвер" будет выдавать те дескрипторы,

которые готовы, например, для операции чтения. По сравнению с традиционным системным вызовом poll() преимущество данного способа состоит в том, что прикладная программа должна сообщить ядру о том, за какими дескрипторами нужно наблюдать, только один раз, в то время как традиционный poll() требует указания набора дескрипторов всякий раз при совершении этого вызова.

В ОС **FreeBSD** и **NetBSD** существует механизм, похожий на описанный в предыдущем абзаце, с той разницей, что вместо псевдоустройства, работа с которым осуществляется с помощью системных вызовов `open()` / `read()` / `write()`, общение прикладной программы с ядром осуществляется при помощи специального системного вызова `kqueue()`.

ОС **Linux** (начиная с ядер серии 2.6) предоставляет альтернативу системному вызову `poll()` – набор системных вызовов `epoll_create()`, `epoll_ctl()`, `epoll_wait()`. Используя их, приложение может создать специальный дескриптор для слежения за другими дескрипторами, добавлять в набор дескрипторов дескрипторы, подлежащие отслеживанию, удалять их из него и ожидать наступления тех или иных событий.

В ОС **Linux** с ядрами серии 2.4 рекомендуется использовать так называемые сигналы "реального времени". Их основное отличие от обычных сигналов стандарта POSIX состоит в том, что они не сливаются. При использовании обычных сигналов POSIX в случае, если до того момента, как сигнал будет доставлен процессу и он сможет его обработать, появится еще один (или более) такой же сигнал, процесс будет уведомлен только один раз. Сигналы "реального времени" образуют очередь, то есть процесс будет уведомлен о каждом событии. Основным недостатком этого механизма является возможность переполнения очереди сигналов; учет этой ситуации приводит к усложнению логики программы. При использовании сигналов "реального времени" рекомендуется получать уведомления синхронно (без использования обработчика сигнала), а не асинхронно (с использованием обработчика сигнала), как это обычно делается. Для синхронного получения уведомлений предназначен системный вызов `sigwaitinfo()`.

В заключение данного раздела приведем классификацию механизмов уведомления процесса о состояниях дескрипторов ввода-вывода. Среди них различают:

- уведомление о ситуации (о готовности как таковой);

- уведомление о событии (об изменении ситуации);
- уведомление о завершении операции.

В англоязычной литературе первый подход часто называют "level-triggered" notification, то есть уведомление "по уровню", а второй – "edge-triggered" notification, то есть уведомление "по фронту". Терминология заимствована из схемотехники – например, можно запрограммировать контроллер прерываний на прерывание по фронту и по уровню.

Для разработчика программ это означает следующее. Если при использовании механизма доставки уведомлений "по фронту" и получении уведомления, например о наличии данных для чтения, процесс, обрабатывая событие, прочитает не все данные, то больше уведомления о наличии данных он не получит (до тех пор, пока не появится новая порция данных). Иными словами, уведомление доставляется только в момент изменения ситуации. При использовании механизма с уведомлением "по уровню" процессу будет сообщено о наличии данных всякий раз, когда он попросит.

Третий подход в данном пособии не описывается; заинтересованного читателя можно отослать к описанию группы функций, реализующих асинхронный ввод-вывод: `aio_read()`, `aio_write()`, `aio_cancel()`, `aio_error()` и др.

Из рассмотренных в данной главе программных механизмов уведомления процесса о состояниях дескрипторов ввода-вывода первому подходу соответствуют `select()`, `poll()`, псевдоустройство `/dev/poll` в ОС **Solaris**, `kqueue()` в ОС **BSD**. Ко второму подходу относятся сигналы и набор вызовов `epoll_create()`, `epoll_ctl()`, `epoll_wait()`. Впрочем, набор `epoll_*()` можно настроить и так, чтобы уведомления происходили "по уровню".

Контрольные вопросы к главе 6

1. Перечислите основные подходы к организации работы с несколькими дескрипторами ввода-вывода одновременно. В чем состоит сущность каждого из них?
2. Какими недостатками обладает подход, основанный на использовании нескольких процессов или нитей?
3. Что значит "неблокирующий ввод-вывод"? Как неблокирующий ввод-вывод можно использовать для организации мультиплексирования потоков ввода-вывода? Каков основной недостаток такого способа мультиплексирования?
4. Для чего предназначены системные вызовы `poll()` и `select()`?
5. Как можно организовать ввод-вывод по сигналам?
6. Если используется набор системных вызовов для работы с сигналами по стандарту **ANSI C**, то каким образом обработчик сигнала `SIGIO` может определить, какой из дескрипторов готов к операции ввода-вывода? Как это же осуществляется при использовании системных вызовов для работы с сигналами по стандарту **POSIX**? Какой стандарт, по вашему мнению, более предпочтителен для использования в программах?
7. Что значит "реентерабельная функция"?

Глава 7.
Примеры программ

В этой главе приведены несколько простых программ, демонстрирующих использование изложенных выше программных механизмов. Примеры размещены в отдельной главе для того, чтобы не перемежать изложение общих принципов и концепций с мелкими деталями. Помимо примеров, здесь представлено много вспомогательного материала по программированию в среде операционных систем семейства Unix. Все примеры были откомпилированы и проверены на работоспособность в операционной системе Linux (дистрибутив RedHat 7.2) с одним из ядер серии 2.4.

7.1. Работа с процессами

Для операционной системы процесс – это наименьшая единица, потребляющая системные ресурсы (процессорное время, оперативную память). Каждый процесс исполняет некоторую программу. Разные процессы могут исполнять одну и ту же программу. В **Unix** процессы связаны друг с другом отношениями "родства". О процессе, который породил другой процесс, говорят как о "родительском процессе" (parent process), а о порожденном – как о "дочернем процессе" (child process) данного процесса. Корень иерархии всех процессов в системе – процесс с именем init, который создается особым образом при запуске системы. Каждый процесс имеет уникальный внутри системы номер, который называется **PID** (аббревиатура от process identifier).

Для прикладной программы доступны перечисленные ниже элементарные операции по управлению процессами (в скобках указаны соответствующие системные вызовы):

- создание процесса (`fork()`)
- завершение исполнения (`_exit()`)
- запуск другой программы (`execve()`)
- ожидание завершения дочернего процесса (`wait()`)
- получение значения своего **PID** (`getpid()`)
- получение значения **PID** родительского процесса (`getppid()`)

Для создания процесса предназначен вызов `fork()`. Все процессы в системе порождаются с помощью этого системного вызова, за

исключением самого первого процесса (**init**), который не может быть создан таким образом по очевидной причине отсутствия процесса, который мог бы это сделать. Вызов выполняет простое действие, а именно: он просто копирует процесс, сделавший этот вызов. Порожденный процесс ничем, кроме значений **PID** и **PPID** (parent **PID**), от родительского процесса не отличается. Если эта операция завершилась успешно, то вызов возвращается сразу в два процесса: родительский и порожденный; при этом первому возвращается **PID** порожденного только что процесса, а второму – 0. Использование кода возврата `fork()` – единственный способ для программы различить, кто родитель, а кто порожденный. Если же процесс создать не удалось, например по причине нехватки памяти или потому, что число процессов в системе превысило бы максимально возможное, то этот вызов возвращается, естественно, только к несостоявшемуся родителю и возвращает константу `-1`. Вызов не имеет параметров.

Завершение процесса осуществляется с помощью вызова `_exit()`. Он выполняет следующие действия: все открытые процессом файлы закрываются, все прямые потомки данного процесса "удочеряются" процессом **init**, родительскому процессу данного процесса посылается сигнал **SIGCHLD**, и, наконец, процесс завершается, освобождая занимаемую им память (однако не всю, смотри ниже про вызов `wait()`). Обычно программисты используют не системный вызов, а "обертку" к нему – библиотечную функцию `exit()`. Эта функция, перед тем как обратиться к системному вызову `_exit()`, вызывает все функции, зарегистрированные с помощью `on_exit()`, `atexit()`, а также сбрасывает все буферы ввода-вывода. Функция `exit()` и системный вызов `_exit()` никогда не возвращаются (по понятной причине). У функции `exit()` имеется один параметр (целое число), посредством которого завершающийся процесс может передать некоторую информацию родительскому процессу. Заметим, что, несмотря на то что этот параметр имеет тип `int`, код возврата дочернего процесса не может превышать значения 255, поскольку, кроме непосредственно кода возврата, это число содержит и другую информацию.

Рассмотрим теперь вызов `wait()`. Этот вызов приостанавливает выполнение текущего процесса до тех пор, пока не завершится хотя бы один из дочерних процессов (или не придет сигнал, который этим процессом не игнорируется). Если к моменту вызова `wait()` хотя бы

один процесс-потомок уже завершился, возврат происходит сразу, без перехода в состояние ожидания.

Зачем нужно ждать завершения потомка? Дело в том, что при завершении процесса он продолжает использовать небольшое количество оперативной памяти; в частности в ней содержится код возврата. Эта память освобождается только тогда, когда родительский процесс делает `wait()`. О процессах, которые завершились, но родительский процесс еще не "ждал" их, иногда говорят как о процессах "зомби". Вызов `wait()` возвращает **PID** завершившегося процесса-потомка. Он имеет один параметр, указатель на целое число, в которое по возвращении из вызова записывается статус завершившегося потомка. Заметим, что этот статус отнюдь не равен значению числа, которое завершившийся потомок передал в качестве параметра функции `exit()`; он содержит дополнительную информацию. Что касается непосредственно кода возврата, то он содержится в младшем байте статуса. Подробности относительно того, какую еще информацию несет статус и как ее оттуда извлекать, смотрите в описании вызова `wait()`, там же описан и вызов `waitpid()`, позволяющий более гибко организовать деятельность по ожиданию завершения потомков; в частности, можно ожидать завершения конкретного процесса-потомка, а также производить ожидание без блокировки.

Возможны, однако, ситуации, когда родительскому процессу не нужно знать код возврата дочернего. В таких ситуациях, чтобы избавиться от необходимости ожидания потомков и связанных с этим усложнений алгоритма программы (помним про слияние сигналов), можно поступить следующим образом. В начале работы нужно определить в качестве реакции на сигнал SIGCHLD "игнорировать". В этом случае после завершения дочернего процесса он уничтожается полностью сразу, без необходимости ожидания со стороны родительского.

Получение **PID** и **PPID** производится с помощью вызовов `getpid()` и `getppid()`, соответственно. Ниже приведен пример, демонстрирующий создание процесса, получение значений **PID** и **PPID**, завершение процесса и ожидание завершения дочернего процесса.

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

pid_t pid;
```

```
int main(void)
{
    printf("\nfork(), exit(), getpid(),
           getppid(), wait() example\n\n");

    printf("\\/ step 1 (potential PARENT): forking ... ");
    fflush(stdout);
/*
без сброса буфера стандартного потока вывода дочерний процесс
тоже выведет эту строчку; конечно, сброс можно осуществить,
выводя в этот поток символ конца строки, '\n'.
*/

    pid = fork();

    if ( pid == -1 ) {
         printf("fork()'ing FAILED\n");
         exit(0);
    };

    if ( pid == 0 ) { // это дочерний процесс
         pid_t my_pid = getpid();
         pid_t pa_pid = getppid();

         printf("\\/ step 1 (CHILD): Hello, world! \
              I am child process.\n");
         printf("\\/ step 2 (CHILD): I am %d and my \
              parent is %d.\n", my_pid, pa_pid);

         printf("\\/ step 3 (CHILD): exiting ...\n");
         sleep(5); // пусть родитель немного подождет exit(123);

    } else {          // это родительский процесс
         pid_t my_pid = getpid();
         pid_t pa_pid = getppid();
         pid_t fork_pid = pid;
         int child_status;

         printf("(PARENT) fork()'ing OK\n");

         printf("\\/ step 2 (PARENT): Hello, world! I \
                  am parent process.\n");
         printf("\\/ step 3 (PARENT): I am %d,
                  my parent is %d and \
                  my child is %d.\n",
                  my_pid, pa_pid, fork_pid);
         printf("\\/ step 4 (PARENT): waiting for \
              child complition ...\n");
```

```
        pid = wait(&child_status);
        if ( pid == -1 ) {
            printf("(PARENT): ooopssss ... \
            wait()'ing FAILED.\n");

        } else {
            printf("(PARENT): wait()'ing OK, \
            child %d exited, \
            exit code is %d.\n",
            pid, WEXITSTATUS(child_status));
/*
WEXITSTATUS(child_status) даст 123, то есть число, возвращенное
дочерним процессом; смотри чуть выше - exit(123) в дочернем
процессе
*/
        };
        printf("\\/ step 5 (PARENT): exiting ...\n");
        exit(0);
    };
}
```

Вызов `fork()` просто копирует процесс, сделавший этот вызов, поэтому функциональность (код) дочернего процесса при этом никак не изменяется. Поскольку потребности пользователей могут быть самым разнообразными, то нужно иметь средство изменения функциональности процесса, то есть запуска в нем на исполнение другой программы. Это осуществляется посредством вызова `execve()`. У этого вызова 3 параметра, первый – указатель на строку, содержащую имя файла, из которого следует взять программу; второй – массив указателей на строки, содержащие аргументы для запускаемой программы и, наконец, третий – массив указателей на строки, содержащие переменные окружения.

Отметим, что в качестве запускаемой программы может выступать сценарий на каком-либо языке описания сценариев, например языке командного интерпретатора bash, языке perl и прочих им подобных. Сценарий в этом случае должен содержать в своей первой строке #!interpreter [arg], где interpreter – это имя файла, содержащего программу, которая будет интерпретировать сценарий, а [arg] – параметры для этого интерпретатора.

Вызов `execve()` меняет содержимое сегментов стека, кода и данных со старого (то, что было у вызывающего процесса) на новое (то, что есть у загружаемой программы) и, таким образом, изменяет программу, которую процесс выполняет. Если требуемую программу запустить

не удалось, то вызов возвращает `-1`. Если же запуск прошел успешно, то вызов не возвращается вообще.

Необходимо отметить, что к описанному системному вызову в **libc** есть семейство оберток, а именно: `execl()`, `execlp()`, `execle()`, `execv()` и `execvp()`. Отличаются они от системного вызова формой передачи параметров и несколько иным поведением (например, могут искать выполнимый файл, если путь к нему указан не полностью, аналогично тому, как это делает командный интерпретатор).

Приведем пример использования вызова `execve()`. Программа "двоит" процесс, дочерний процесс запускает программу `ps` с параметрами `aux`, родительский просто ожидает завершения потомка. Обратите внимание, что первый (у которого номер 0) параметр для программы `ps` – это имя ее самой. В **Unix** имеется такая договоренность: первым параметром передавать имя программы, однако ответственными за соблюдением этой договоренности являются процессы, а не операционная система. Если первым параметром сделать `aux`, то должного результата не получится, поскольку программа `ps`, как и все программы в Unix, полагает, что параметры для нее начинаются с `argv[1]`, а не с `argv[0]`.

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

pid_t pid;

int main(void)
{
    printf("\nfork()-exec() example\n\n");

    pid = fork();

    if ( pid == -1 ) {
         perror("--PARENT-- fork()");
         exit(0);
    };

    if ( pid == 0 ) { // дочерний процесс
         char * program = "/bin/ps";
         char * argv[] = {"/bin/ps","aux", NULL};
         char * envp[] = {NULL};
```

```
        int ret;
        ret = execve(program, argv, envp);
        if ( ret == -1 ) {
            perror("--CHILD-- execve()");
            exit(1);
        };

    } else { // родительский процесс
        int child_status;

        pid = wait(&child_status);
        if ( pid == -1 ) {
            perror("--PARENT-- wait()");
        };

        exit(0);
    }

    return 0;
}
```

В заключение этого раздела рассмотрим так называемые процессы-демоны (daemons). Демон – это процесс, работающий в фоновом режиме и не имеющий управляющего терминала. Такие процессы не могут читать со стандартного устройства ввода и выводить на стандартное устройство вывода. Как правило, все программы-серверы являются демонами, например ftp-сервер (ftpd), http-сервер (httpd), dns-сервер (named), почтовые серверы (smtpd, imapd). Чтобы стать демоном, программа должна закрыть все стандартные каналы ввода-вывода и создать так называемую "сессию". Последнее действие совершается с помощью системного вызова `setsid()`. Ниже приведен короткий пример того, как это делается.

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

pid_t pid;

int main(void)
{
    printf(" how to become a daemon\n");

    // создаем новый процесс
    pid = fork();
```

```
    if ( pid == -1 ) {
         perror("can't fork()");
         exit(1);
    };

    if ( pid > 0 ) { // родительский процесс

         // здесь, например, можно проверить
         // тем или иным образом,
         // что дочерний процесс успешно проинициализировался

         exit(0); // родительский процесс завершается

    } else { // pid == 0, дочерний процесс

         close(STDIN_FILENO);
         close(STDOUT_FILENO);
         close(STDERR_FILENO);
         setsid();
         // после закрытия стандартных устройств ввода-вывода
         // и создания сессии дочерний процесс является "демоном"
         while(1) {
              sleep(60);
         };
         // здесь можно запустить на исполнение
         // какую-то другую программу
    }
}
```

Пояснения к примеру "демонизации". В ОС семейства Unix процессы могут объединяться в группы. Это сделано для того, чтобы можно было послать сигнал не отдельному процессу, а группе. Все процессы, входящие в группу, имеют одинаковый идентификатор группы. У группы процессов имеется лидер группы. Это такой процесс, идентификатор группы которого совпадает с идентификатором этого процесса. В свою очередь группы процессов могут объединяться в так называемые "сеансы" или "сессии". Аналогично группам, в сеанс входят все процессы, у которых идентификатор сеанса одинаков. Один из процессов, входящих в сеанс, является его лидером. Сеанс может иметь управляющий терминал, что используется для организации интерактивного взаимодействия с пользователем. Создание сеанса (сессии) производится, как уже упоминалось, с помощью системного вызова `setsid()`. Создать сессию не может процесс, являющийся лидером группы.

Именно по этой причине в приведенном примере перед "демонизацией" используется `fork()` - когда родительский процесс завершается, дочерний становится потомком процесса init, что гарантирует то, что этот потомок после этого не является лидером группы. После вызова `setsid()`, то есть после создания сеанса, данный процесс становится лидером этого нового сеанса и лидером группы.

Замечание по поводу закрытия стандартных потоков ввода-вывода. Иногда рекомендуют не закрывать их, а перенаправить все эти потоки в псевдоустройство `/dev/null`. Это делается из опасения, что какая-нибудь функция из используемых программой библиотек (или же сама программа, вследствие неаккуратности программиста) может выводить данные в стандартный поток вывода или в стандартный поток диагностического вывода. Если речь идет о серверах, то потоки с дескрипторами 1 и 2 могут соответствовать сокетам, поэтому запись в эти потоки "случайной" информации будет нарушением протокола взаимодействия. Если же мы перенаправим эти потоки, то этим самым мы гарантируем, что потоки с дескрипторами 0, 1, 2 остаются открытыми и поэтому никакой сокет не будет иметь такой дескриптор.

В заключение этого раздела отметим, что в стандартной библиотеке имеется функция `daemon()`, которая выполняет "демонизацию", при этом имеется возможность указать, что нужно сделать с дескрипторами 0, 1 и 2 - просто закрыть их или же перенаправить в `/dev/null`.

7.2. Использование select() и poll()

В этом разделе рассматривается вопрос о том, как организовать работу с более чем одним дескриптором потока ввода-вывода при помощи вызовов `select()` и `poll()`. Эти системные вызовы используются для синхронного мультиплексирования ввода-вывода. Рассмотрим сначала вызов `select()`.

Для удобства его использования имеется ряд вспомогательных макроопределений (`FD_CLR`, `FD_ISSET`, `FD_SET`, `FD_ZERO`), которые применяются для манипуляций так называемыми наборами дескрипторов. Набор дескрипторов – это битовый массив, каждый бит которого определяет, входит ли данный дескриптор в набор или не входит. Набор

дескрипторов представлен типом `fd_set`; иными словами, в программе набор дескрипторов – это переменная обозначенного типа.

Первые три макроопределения (`FD_CLR`, `FD_ISSET`, `FD_SET`) имеют два параметра, дескриптор и указатель на набор дескрипторов. Они производят следующие действия: первый исключает данный дескриптор из данного набора, второй проверяет, входит ли данный дескриптор в данный набор и, наконец, третий включает данный дескриптор в данный набор. Последнее макроопределение (`FD_ZERO`) имеет один параметр – указатель на набор дескрипторов; оно очищает данный набор, то есть удаляет из него все дескрипторы.

Прототип вызова `select()` имеет следующий вид:

```
int select(
            int n,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout
);
```

Вызов `select()` имеет пять параметров; второй, третий и четвертый – это наборы дескрипторов (точнее, указатели на них), за которыми `select()` будет наблюдать. Дескрипторы из первого набора отслеживаются на предмет наличия данных для чтения, из второго – на предмет возможности немедленной записи и из третьего – на предмет возникновения исключительных ситуаций. Первый параметр – целое число, которое должно быть на единицу больше максимального (по значению) файлового дескриптора в этих трех наборах, а пятый – это указатель на структуру, содержащую максимальное время ожидания изменения состояния дескрипторов в наборах (так называемый "тайм-аут"). Если это время равно 0, то `select()` возвращается немедленно; если же сам указатель равен `NULL`, то тайм-аут равен бесконечности. Вызов возвращает число дескрипторов, готовых к операциям ввода-вывода, то есть суммарное число дескрипторов, оставшихся в трех наборах.

Перед тем как воспользоваться вызовом select(), следует подготовить нужные нам наборы. Если мы, например, не хотим проверять возможность немедленной записи, то в качестве указателя на соответствующий набор можно передать константу NULL. Подготовить означает включить (используйте FD_SET) в тот или иной набор все дескрипторы, состояние которых мы желаем отслеживать. После возврата из вызова

в наборе останутся только те дескрипторы, которые готовы к той или иной операции ввода-вывода.

Для выявления того, какие дескрипторы остались в наборе, а значит, готовы к той или иной операции, используйте FD_ISSET. Не забывайте отслеживать максимальный по величине дескриптор: полагаться на то, что самый "большой" дескриптор – это тот, который был возвращен самым последним вызовом open() или socket(), не следует, потому что некоторые из созданных ранее дескрипторов уже могли быть закрыты и их номера могли быть повторно использованы для вновь создаваемых дескрипторов.

Пример использования `select()` имеется в страничке руководства программиста, посвященной этому системному вызову (man select), а также в разделе 8.4, в котором приведен пример программы-сервера.

Остановимся еще на том, какой смысл имеет то обстоятельство, что какой-либо дескриптор остался в наборе после возврата `select()`. Первый набор – дескрипторы, проверяемые на наличие данных для чтения. Для терминалов и обычных сокетов наличие дескриптора в этом наборе после возврата `select()` означает, собственно, то, что имеются данные для чтения (вызов `read()` или `recv()` не заблокируется).

В отличие от них, для сокета, который находится в режиме прослушивания (см. далее), готовность дескриптора к чтению означает, что была попытка установления соединения (вызов `accept()` не заблокируется). Второй набор – дескрипторы, проверяемые на возможность записи данных без блокировки. Отметим, что для сокетов это можно использовать для определения того, завершился ли вызов `connect()` или нет.

Перейдем теперь к рассмотрению вызова `poll()`. Как уже говорилось, он предназначен для достижения тех же целей, что и вызов `select()`. Отличие состоит в том, что информация о том, за какими дескрипторами надо следить и какие из дескрипторов готовы к записи-чтению, организована иным способом. Вызов имеет три параметра: первый представляет собой массив некоторых структур, второй – длина массива, то есть количество этих структур (беззнаковое целое), третий – тайм-аут в миллисекундах (знаковое целое). Знаковое потому, что любое отрицательное значение используется для указания бесконечного тайм-аута. При успешном завершении вызов возвращает количество

дескрипторов, готовых к операциям ввода-вывода или имеющих какую-то исключительную ситуацию (ошибку): 0 – если произошел тайм-аут, а при ошибке, как обычно, константу `-1` с установлением соответствующего значения переменной `errno`.

Элементами массива (первого параметра вызова) являются структуры следующего вида:

```
struct pollfd {
      int fd;           /* file descriptor */
      short events;     /* requested events */
      short revents;    /* returned events */
};
```

Первый элемент структуры – собственно сам файловый дескриптор, второй – битовая маска событий, которые подлежат отслеживанию для этого дескриптора, третий – какие события реально произошли (тоже битовая маска). Второй элемент является входным параметром, а третий – выходным. В качестве событий для отслеживания можно указывать `POLLIN` (есть данные для чтения), `POLLPRI` (есть срочные данные для чтения), `POLLOUT` (запись не будет заблокирована); в качестве выходных событий могут встретиться три упомянутые (из числа тех, которые были запрошены, естественно), а также дополнительно `POLLERR` (ошибка), `POLLHUP` (связь разорвана), `POLLINVAL` (неправильный запрос, открытого файла с таким дескриптором нет). Запрос на отслеживание нескольких типов событий формируется из упомянутых констант посредством операции побитового ИЛИ. Например, `POLLIN | POLLPRI` – будем отслеживать, когда появятся данные для чтения или срочные данные для чтения.

Далее приводится пример использования вызова `poll()`.

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

struct pollfd stdinfd = {0,POLLIN,0};
/*
     одна структура struct pollfd;
          первое поле структуры - НУЛЬ, значит, отслеживаем де-
          скриптор НУЛЬ (стандартное устройсто ввода),
          второе поле структуры означает, что будем отслеживать
```

```
          события, состоящие в том, что имеются данные для чте-
          ния;
          третье поле - выходное, заполнять его чем-то особенным
          не нужно,
          поэтому - просто произвольное число, здесь - нуль.
*/

int poll_ret;
char buf[1024];
int timeout = 7000; /* 7s */
int delta = 1000;

int main(void)
{
    while (1) {

         printf("U have %d ms, type \
              sumthin (end with /enter/) ... ",
              timeout);

         fflush(stdout);

         poll_ret = poll(&stdinfd, 1, timeout);

         if ( poll_ret == 0 ) {
              printf("TIMED OUT, nuthin entered, bye :-P\n");
              break;
         };

         if ( poll_ret == -1 ) {
              perror("poll");
              break;
         };

         if ( poll_ret > 0) {
              scanf("%s", buf);
              printf("u entered <%s>, cooool, \
              lets do it again!\n", buf);
         };

         timeout -= delta;
         if ( timeout == delta )
              delta = delta/10;
    };

    return 0;
}
```

Программа работает с одним дескриптором (0, стандартный ввод). Просит ввести что-нибудь; если в течение заданного тайм-аута ничего не вводилось, завершается. Тайм-аут все время уменьшается. Для терминала наличие данных для чтения означает, что была нажата клавиша Enter. Программа не проверяет, какое именно событие произошло (не анализирует поле `stdinfd[0].revents`).

7.3. Работа с сигналами

Сигнал – это программное прерывание, доставляемое процессу. Как уже отмечалось, "доставка прерывания" означает, что при возникновении некоторого события операционная система вызывает заданный процессом обработчик этого события. Событие может быть вызвано как деятельностью самого процесса, например обращением к ячейке памяти, к которой процесс не имеет доступа, так и внешними по отношению к процессу причинами, обусловленными действиями как других процессов, так и пользователей.

Процесс может отреагировать на сигнал тремя "способами", а именно: он может отреагировать стандартно (вызывается стандартный обработчик), может проигнорировать сигнал (никакой обработчик не вызывается) и, наконец, он может реагировать произвольным образом (вызывается нестандартный обработчик, то есть обработчик, определенный программистом, разрабатывающим программу). Однако не для всех сигналов у процессов имеется такая свобода выбора. Для некоторых сигналов реакция фиксирована (`SIGKILL`, `SIGSTOP`, `SIGCONT`). Такие сигналы нельзя перехватывать (задавать собственный обработчик для них) и их также нельзя игнорировать. Кроме того, есть сигналы, игнорирование которых возможно, но не рекомендуется, ибо приводит к неопределенному поведению (например, `SIGSEGV`, `SIGFPE`, `SIGILL`).

Рассмотрим примитивы работы с сигналами, определенные стандартом POSIX. Процесс может определить реакцию на сигнал, может послать сигнал какому-либо процессу (в частности, самому себе), может ждать прихода сигнала, а может блокировать сигналы и деблокировать их.

Для определения реакции на сигнал (установки обработчика) используется системный вызов `sigaction()`. Напомним, что речь идет именно о стандарте POSIX; в стандарте ANSI C для этих целей используется вызов `signal()`, который мы рассматривать не будем.

Вызов `sigaction()` имеет 3 параметра: первый задает номер сигнала, на который устанавливается реакция, второй задает, собственно, эту реакцию в виде указателя на некоторую структуру, третий параметр (указатель такого же типа) используется для возвращения информации о старом обработчике. Сигнал, на который определяется реакция, может быть любым допустимым в данной системе сигналом, кроме `SIGKILL`, `SIGSTOP` и `SIGCONT`. Структура, в которой содержится информация об обработчике сигнала, содержит 5 полей:

```
struct sigaction {
         void (*sa_handler)(int);
         void (*sa_sigaction)(int, siginfo_t *, void *);
         sigset_t sa_mask;
         int sa_flags;
         void (*sa_restorer)(void);
}
```

Последний элемент считается устаревшим, использовать его не следует. Первый элемент – указатель на функцию-обработчик сигнала; он может также иметь значения `SIG_DFL` (установить на данный сигнал реакцию по умолчанию) и `SIG_IGN` (игнорировать данный сигнал). Третий элемент задает набор сигналов, которые будут блокированы во время работы обработчика. Блокирован будет (по умолчанию) также сигнал, который обрабатывается. Четвертый элемент – набор флагов, модифицирующих поведение подсистем ОС и libc, ответственных за управление сигналами. Он формируется посредством операции побитового "ИЛИ" из следующих возможных флагов.

- Флаг `SA_NOCLDSTOP` означает, что не надо посылать сигнал `SIGCHLD`, когда процесс-потомок останавливается.
- Флаг `SA_ONESHOT` (`SA_RESETHAND`) означает, что после того, как обработчик сигнала один раз проделает свою работу, в качестве реакции на данный исгнал устанавливается реакция «по умолчанию»; интыми словами, обработчик, определенный пользователем, будет вызван только один раз, в дальнейшем будет вызываться обработчик по умолчанию.
- Флаг `SA_RESTART` означает, что если во время исполнения системных вызовов приходит сигнал и его обработчик нормально возвращается, то вызов производится повторно; в случае, если этот флаг не установлен, прерванный сигналом системный вызов завершается с ошибкой (`errno` при этом принимает значение `EINTR`).

- Флаг SA_NOMASK означает, что сигнал не будет блокироваться во время работы его обработчика.
- Флаг SA_SIGINFO говорит о том, что обработчик сигнала будет принимать не один параметр, а три.

Важно отметить, что если флаг SA_SIGINFO установлен, то в структуре struct sigaction перед вызовом sigaction() необходимо вместо элемента sa_handler заполнять элемент sa_sigaction. В этом случае обработчику сигнала будет передан указатель на некоторую структуру. Эта структура содержит массу информации, в частности номер сигнала, значение errno, **PID** процесса, который послал сигнал, номер файлового дескриптора (для случая сигнала SIGIO).

Полное описание элементов этой структуры смотрите в описании sigaction(). Здесь нас больше всего интересует то, что для сигнала SIGIO при условии соблюдения описанных требований обработчику этого сигнала будет передан номер файлового дескриптора, который оказался готовым к проведению операции ввода-вывода.

Для посылки сигналов используется вызов kill(); отметим, что он одинаков в обоих упомянутых стандартах. Вызов имеет два параметра. Первый задает получателя сигнала. Второй представляет собой номер сигнала, который нужно послать. При успешном завершении kill() возвращает 0, а при ошибке, как обычно, константу −1.

Перед тем как продолжить рассмотрение действий по отношению к сигналам, опишем вспомогательные функции, определенные стандартом POSIX и предназначенные для манипуляций так называемыми наборами сигналов. Всего их 5, называются они sigemptyset(), sigfillset(), sigaddset(), sigdelset(), sigismember(). Первые две имеют один параметр – указатель на набор сигналов; остальные функции имеют два параметра, первый – такой же указатель, второй – номер сигнала.

Названия этих функций говорят сами за себя: sigemptyset() очищает заданный набор сигналов, то есть удаляет из него все сигналы; sigfillset() включает в заданный набор все возможные сигналы; sigaddset() добавляет заданный сигнал в заданный набор; sigdelset() удаляет заданный сигнал из заданного набора; sigismember() проверяет, входит ли заданный сигнал в заданный набор. Все функции, кроме последней, возвращают –1 при ошибке и 0

при успешном завершении. Последняя возвращает 1, если сигнал имеется в наборе, 0 – если не имеется и –1 при ошибке. Три системных вызова для работы с сигналами по стандарту POSIX имеют хотя бы один параметр, представляющий собой указатель на набор сигналов. Для подготовки набора сигналов перед передачей их этим вызовам, а также для интерпретации содержимого набора после возврата из них как раз и предназначены описанные только что функции.

Продолжим теперь рассмотрение действий, которые можно осуществлять по отношению к сигналам. Сигналы можно блокировать и деблокировать. Когда сигнал заблокирован, его обработчик не вызывается, даже если имеется сигнал, требующий обработки. Сигнал как бы ожидает обработки (на английском языке такой сигнал называется "pending signal" – "сигнал, ожидающий обработки", "висячий сигнал"). Обработчик сигнала будет вызван сразу, как только процесс разблокирует этот сигнал.

Блокировка сигнала, по сути, означает временное игнорирование сигнала; разница в том, что если процесс никогда не хочет получать данный сигнал, он его игнорирует, а **ОС** при возникновении данного сигнала для данного процесса его просто аннулирует; если же для процесса нежелательно получать тот или иной сигнал лишь в течение некоторых промежутков времени, то он его блокирует, а **ОС** сохраняет информацию о наличии сигнала для данного процесса и доставляет его (сигнал) процессу, когда последний разблокирует этот сигнал.

Совокупность всех сигналов, которые в данный момент времени заблокированы, – маска сигналов. У каждого процесса, естественно, своя маска сигналов; при создании процесса она наследуется от родительского процесса. Для манипуляций с маской сигналов предназначен системный вызов `sigprocmask()`. Не следует путать их с описанными выше манипуляциями над наборами сигналов, которые ничего не блокируют и не деблокируют.

Для чего нужна блокировка сигналов? При наличии участков кода программы, которые получают управление асинхронно, вне связи с тем, что делает процесс в данный момент, могут возникать несколько специфичные проблемы. Такие проблемы возникают практически всегда, когда два или более участка кода программы, из которых по крайней мере один получает управление асинхронно, используют при работе некоторый общий ресурс.

Простейшим примером такого общего ресурса может быть глобальная переменная. Следует заметить, однако, что это может быть не всякая переменная. Если любой доступ к переменной осуществляется за одну машинную инструкцию, то описываемых далее проблем не может возникнуть в принципе, поскольку процессор по физическим причинам не может приостановить выполнение инструкции и всегда ее выполняет до конца. Во время выполнения инструкции процессор даже не проверяет, не поступил ли сигнал прерывания от какого-либо внешнего устройства; следовательно, **ОС**, код которой получает управление именно по прерываниям (системные вызовы не в счет, это вещь синхронная, инициируются они самим процессом), не может ни переключить контексты процессов, ни вызвать обработчик сигнала.

Как известно, участки кода, которые имеют дело с общим ресурсом произвольной природы, называются критическими секциями. Если основная программа и обработчики сигналов имеют общие ресурсы, то на время, пока основная программа с ними работает (читает, пишет – не важно), соответствующие сигналы в большинстве случаев должны быть заблокированы. Кроме того, сигнал может прийти во время исполнения обработчика этого самого сигнала. Поэтому на время работы обработчика сигнала этот сигнал также желательно блокировать, в противном случае возможно получение вложенного вызова обработчика, что может привести к различного рода неприятным последствиям.

Еще пример. Допустим, в программе нужно выполнить некое действие только в том случае, если сигнал *не* пришел. Обработчик сигнала устанавливает флаг. В основной программе вы проверяете этот флаг и, если он не установлен, выполняете требуемые действия. Такой алгоритм ненадежен! Дело в том, что сигнал может прийти в момент времени сразу после проверки флага, но до начала выполнения нужных действий. Можно возразить, что так не может быть, что между проверкой флага и началом действий нет никакого промежутка.

На самом деле это не так. Эти две операции, как правило, осуществляются за несколько машинных инструкций. После любой из них управление может получить обработчик. Может случиться так, что управление он получит тогда, когда основная программа только что проверила флаг, но не начала последующих действий. В результате сигнал пришел, а вы выполните действия, которые намеревались выполнить только в том случае, если сигнал не придет. Самое неприятное, что так будет получаться не всегда, а время от времени, причем, скорее всего, очень редко (возможно, никогда), то есть возникающую ошибку будет очень

трудно (читай – практически невозможно) воспроизвести по собственному желанию. Правильный способ проверки флага состоит в том, чтобы на время проверки блокировать сигнал, обслуживаемый обработчиком, устанавливающим этот флаг.

Вызов `sigprocmask()` имеет три параметра. Первый (целое число) задает, что надо делать с маской сигналов. Второй параметр (входной) – указатель на набор сигналов, который задает требуемую маску сигналов. Третий параметр (выходной) – также указатель на набор сигналов, используется для возвращения старой маски сигналов, то есть той, которую процесс имел в момент обращения к `sigprocmask()`. Первый параметр может принимать значения `SIG_BLOCK`, `SIG_UNBLOCK`, `SIG_SETMASK`. В первом случае к маске сигналов добавляется набор сигналов, задаваемый вторым параметром. Во втором случае из маски сигналов удаляются сигналы, входящие в набор, задаваемый вторым параметром (то есть эти сигналы деблокируются). Наконец, в третьем случае маска сигналов заменяется указанным набором. Если неинтересно, какая была маска до вызова, в качестве третьего параметра передаем `NULL`. Если не желаем ничего менять, а просто нужно знать, какая сейчас маска, передаем в качестве второго параметра `NULL`. Вызов возвращает 0, если все нормально, –1 при ошибке. Если вызов приводит к разблокированию сигналов, которые уже ожидали обработки, то он возвращается только после того, как будет вызван, по крайней мере, один из обработчиков этих сигналов. Порядок вызова обработчиков никак не определяется; если это важно, разблокируйте по одному сигналу за один вызов.

Нам осталось рассмотреть операцию ожидания прихода сигнала. Ее можно производить с помощью двух вызовов: `pause()` и `sigsuspend()`. Первый определен также в стандарте ANSI C. Второй, однако, более гибок в использовании. Вызов `pause()` переводит вызывающий процесс в состояние сна до тех пор, пока не придет какой-нибудь сигнал. Параметров он не имеет, возвращает всегда `-1`.

Сделаем одно замечание по поводу использования этого вызова. Оно наиболее надежно в случае, если все манипуляции с какой-то группой данных выполняются в обработчиках сигналов, а основная программа ничего, кроме циклического обращения к `pause()`, не делает. Если же процесс обработки данных по каким-то причинам необходимо разделить между обработчиком сигнала и основной программой, то использование этого вызова становится ненадежным.

Стандарт POSIX, однако, предлагает другое средство для ожидания сигнала, а именно системный вызов `sigsuspend()`. Вызов имеет один параметр – указатель на набор сигналов. Он переводит процесс в состояние сна до тех пор, пока не придет один из сигналов, не входящий в указанный набор. Сигналы, не входящие в этот набор, на время исполнения вызова блокированы (то есть вызов `sigsuspend()` изменяет маску сигналов). После возврата маска сигналов восстанавливается. Фактически данный вызов позволяет ожидать определенные сигналы, в то время как другие сигналы будут обрабатываться соответствующими обработчиками.

Ниже приведен исходный текст простой (и практически бесполезной) программы, демонстрирующей блокировку и деблокировку сигналов, а также использование примитивов для работы с наборами сигналов.

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

extern const char * const sys_siglist[];

void print_blocked(void)
{
     int nblocked = 0;
     int sig;
     sigset_t blocked;
     int ret;

     // get signal mask
     ret = sigprocmask(SIG_BLOCK, NULL, &blocked);
     if ( ret == -1 ) {
          perror("sigprocmask()");
          exit(1);
     };

     for ( sig = 0; sig < NSIG; sig++ ) {
          if ( sys_siglist[sig] ) {

               if ( sigismember(&blocked, sig) ) {

                    printf("#%d (%s)\n",
                           sig,
                           sys_siglist[sig]);
                    nblocked++;
               };
          };
     };
```

```
    if ( nblocked )
        printf(" total %d signal(s) blocked } \n\n",
               nblocked);
    else
        printf(" none } \n\n");
}


sigset_t blocked_orig;
sigset_t blocked_new;
int ret;


int main(void)
{
    printf("\nsigprocmask() example\n\
              ~~~~~~~~~~~~~~~~~~~~\n\n");

    printf(" { originally blocked signals:\n");
    print_blocked();

    // формируем набор сигналов,
    // содержащий SIGINT, SIGTERM, SIGSEGV
    ret = sigemptyset(&blocked_new);
    ret = sigaddset(&blocked_new, SIGINT);
    ret = sigaddset(&blocked_new, SIGTERM);
    ret = sigaddset(&blocked_new, SIGSEGV);

    // блокируем SIGINT, SIGTERM, SIGSEGV,
    // сохраняем первоначальную маску сигналов
    ret = sigprocmask(SIG_BLOCK, &blocked_new, &blocked_orig);

    if ( ret == -1 ) {
        perror("sigprocmask()");
        exit(1);
    };

    printf(" { currently blocked signals:\n");
    print_blocked();

    ret = sigdelset(&blocked_new, SIGINT);
    ret = sigdelset(&blocked_new, SIGTERM);
    // SIGSEGV все еще в наборе
    // разблокируем SIGSEGV, сохраняем маску
    ret = sigprocmask(SIG_UNBLOCK, &blocked_new, NULL);

    if ( ret == -1 ) {
        perror("sigprocmask()");
        exit(1);
```

```
    };

    printf(" { currently blocked signals:\n");
    print_blocked();

    // восстанавливаем первоначальную маску сигналов
    ret = sigprocmask(SIG_SETMASK, &blocked_orig, NULL);

    if ( ret == -1 ) {
        perror("sigprocmask()");
        exit(1);
    };

    return 0;
}
```

В программе определена одна функция (помимо `main()`), которая выводит на стандартное устройство вывода список заблокированных сигналов. Главная функция распечатывает с ее помощью этот список, затем блокирует сигналы `SIGINT`, `SIGTERM`, `SIGSEGV` (одновременно запоминая текущую маску сигналов), снова печатает список, наконец деблокирует `SIGSEGV` и опять распечатывает список. Перед завершением программа восстанавливает первоначальную маску сигналов.

Далее приведен пример, демонстрирующий организацию чтения данных со стандартного устройства ввода с использованием сигнала `SIGIO`.

```
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <string.h>

void io_handler(int sig, siginfo_t* sinfo, void * add_info);
struct sigaction io_action;
struct sigaction io_action_old;
sigset_t blocked;

int main(void)
{
```

```
    printf("\nsigaction() (interrupt i/o) exersise\n");

    if ( fcntl(STDIN_FILENO, F_SETFL, O_ASYNC) == -1 ) {
        perror("fcntl/F_SETFL");
        exit(1);
    };

    if ( fcntl(STDIN_FILENO, F_SETSIG, SIGIO) == -1 ) {
        perror("fcntl/F_SETSIG");
        exit(1);
    };

    if ( fcntl(STDIN_FILENO, F_SETOWN, getpid() ) ) {
        perror("fcntl/F_SETOWN");
        exit(1);
    };

    /* установить обработчик на SIGIO */
    io_action.sa_handler = NULL;
    io_action.sa_sigaction = io_handler;
    io_action.sa_flags = SA_SIGINFO;
    io_action.sa_restorer = NULL;

    if (sigaction(SIGPOLL, &io_action, &io_action_old) == -1) {
        perror("sigaction()");
        exit(1);
    };

    /* работаем */
    sigemptyset(&blocked);
    while (1)
        sigsuspend(&blocked);
}


char in_buf[256] = "";
char out_buf[256] = "";


/* обработчик сигналов */
void io_handler(int sig, siginfo_t* sinfo, void * add_info)
{
    int nbytes, r, fd;

    if ( sig != SIGIO)
        return;
    if (sinfo->si_signo != SIGIO )
        return;

    printf("\nSignal info:\n~~~~~~~~~~~~\n");
```

```
printf("numb:\t %d (%d)\n", sinfo->si_signo, sig);
printf("errn:\t %d\n", sinfo->si_errno);
printf("code:\t %d ", sinfo->si_code);


switch ( sinfo->si_code ) {
case POLL_IN:
     printf("(POLL_IN - \
            data input available)\n");
break;
case POLL_OUT:
     printf("(POLL_OUT - \
             output buffers availble)\n");
break;
case POLL_PRI:
     printf("(POLL_PRI - \
            urgent input data available)\n");
break;
case POLL_MSG:
     printf("(POLL_MSG - \
            input message available)\n");
break;
case POLL_ERR:
     printf("(POLL_ERR - \
            i/o error)\n");
break;
case POLL_HUP:
     printf("(POLL_HUP - \
            device disconnected)\n");
break;
default:
     printf("???\n");
};

printf("chan:\t %d\n", sinfo->si_fd);

if ( sinfo->si_code == POLL_IN) {

     fd = sinfo->si_fd;
     r = ioctl(fd, FIONREAD, &nbytes);

     if ( r == -1 ) {
         perror("ioctl()");
     };

     r = read(fd, in_buf, nbytes);

     if ( r == -1 ) {
         perror("read()");
     };
```

```
            sprintf(out_buf, "data:\t %s", in_buf);
            printf("%s", out_buf);
            memset(in_buf, 0, 256);
            memset(out_buf, 0, 256);
        };
}
```

Программа выполняет следующие действия. С помощью вызова `fcntl()` устанавливает асинхронный режим для стандартного устройства ввода. С его же помощью производится некоторое действие, без которого в поле `sinfo->si_fd` не записывается номер файлового дескриптора, а именно это наиболее полезно. Замечание: `F_SETSIG` определено только тогда, когда определено `_GNU_SOURCE`. Еще один вызов `fcntl()` нужен для определения получателя сигнала `SIGIO`, а иначе он (сигнал), скорее всего, будет послан не нашему процессу, а какому-то другому, например командному интерпретатору, из которого мы запустили эту программу. Далее заполняется структура `sigaction`; здесь важно в поле `flags` установить флаг `SA_SIGINFO`. После этого с помощью вызова `sigaction()` устанавливается обработчик сигнала `SIGIO`. Затем программа переходит в бесконечный цикл, в котором с помощью вызова `sigsuspend()` ожидает прихода сигнала; во время ожидания все сигналы разблокированы. Обработчик сигнала выводит информацию о сигнале (номер, код события, номер дескриптора) и, если событие состоит в том, что готовы данные, читает их и выводит на экран.

Контрольные вопросы к главе 7

1. Перечислите основные системные вызовы для работы с процессами.
2. Для чего предназначен системный вызов fork()?
3. Почему родительский процесс должен ожидать завершения дочернего процесса? С помощью какого системного вызова это осуществляется?
4. С помощью какого системного вызова процесс может запустить другую программу?
5. Что такое "процесс-демон"?
6. Напишите прототип функции select(). Поясните назначение всех ее параметров. Что возвращает эта функция?
7. Проделайте то же самое для функции poll().
8. Что такое "обработчик сигнала"?
9. На какой сигнал нужно установить обработчик для организации асинхронного ввода-вывода?
10. Что такое "маска сигналов"? Как ее можно изменить?
11. Для чего в некоторых случаях необходимо блокирование сигналов? Приведите пример, где такая необходимость возникает.

Глава 8.
Технология клиент-сервер

8.1. Архитектура "клиент-сервер"

Большинство сетевых приложений строится на основе архитектуры, называемой архитектурой "клиент-сервер". При таком подходе задачи по обработке, хранению, передаче данных разделены между несколькими программами. Обычно одна из них является так называемым сервером. Эта программа предоставляет некоторую услугу многочисленным клиентам. Клиент – это программа, которую обслуживает программа-сервер.

В семействе протоколов **TCP/IP** имеются два протокола транспортного уровня – **TCP** и **UDP**. Первый из них является потоково-ориентированным и гарантирует доставку пакетов. Второй является датаграммно-ориентированным и не гарантирует надежную доставку пакетов. Использование протокола **TCP** подразумевает существование так называемого соединения между клиентом и сервером. Вследствие надежности этого протокола именно он используется в большинстве случаев при проектировании сетевых прикладных программ.

Назовем сеансом промежуток времени, в течение которого существует соединение (подразумеваем, что используется транспортный протокол **TCP**) между клиентом и сервером. В зависимости от длительности сеанса (времени существования соединения) к серверу предъявляются разные требования. Если он короткий (по человеческим меркам, например, меньше одной секунды), то сервер допустимо написать таким образом, чтобы в каждый момент времени существовало только одно соединение с каким-то клиентом. Такой сервер будем называть последовательным сервером, так как клиентов он обслуживает по очереди, последовательно. Безусловно, при интенсивной загрузке такого рода сервера (много клиентов, пытающихся установить соединение), несмотря на малое время обслуживания, клиенты будут ожидать соединения в очереди, и время этого ожидания будет, естественно, больше, чем время обслуживания.

Если же природа сервера (оказываемой им услуги) такова, что требует значительного времени (минуты, часы, дни и более), то вероятность одновременного появления многих клиентов возрастает, и подход, при котором все клиенты обслуживаются по очереди (пока не закончен

сеанс с одним клиентом, не может быть начат сеанс с другим клиентом), становится неразумным. Назовем параллельным сервером такой сервер, для которого в каждый момент времени может существовать более одного соединения с клиентами. Вопросу о необходимости организации мультиплексирования ввода-вывода, возникающей в таких случаях, были посвящены две предыдущих главы.

Примерами услуг, для оказания которых можно использовать простые последовательные сервера, могут быть, например, служба выдачи текущего времени, эхо-служба и т. п. Большинство же услуг требует более или менее продолжительного сеанса, поэтому все сервера, их предоставляющие, как правило, являются параллельными. В качестве примера можно упомянуть различного рода файловые сервера (http, ftp), сервера, предоставляющие доступ к машине как к таковой (telnet, ssh) и пр.

Кроме длительности сеанса, при разработке сервера следует также учитывать:

- ❖ длительность выполнения одиночного запроса, вне зависимости от возможного количества запросов в сеансе
- ❖ архитектуру вычислительной системы (один процессор или много)

Так, если природа оказываемой сервером услуги такова, что выполнение единичного запроса от клиента может потребовать значительного времени и/или в вычислительной машине имеется более одного процессора, то разумнее использовать многопроцессный (многонитевый) подход или его комбинацию с мультиплексированием в каждом из процессов (нитей).

В противном случае (запросы всегда отрабатываются быстро и имеется только один процессор) более эффективным может оказаться однопроцессный подход с мультиплексированием. См. также по этому поводу конец раздела 6.1 и введение к разделу 6.2.

При этом заранее очевидно, что в случае файловых серверов типа HTTP или FTP некоторые запросы сервер будет отрабатывать долго (передача большого файла), поэтому, кроме собственно параллельности, они еще должны обеспечить "справедливое" обслуживание клиентов, что наиболее легко достигается при использовании многопроцессного

(многонитевого) подхода Справедливое обслуживание клиентов для сервисов с долгими запросами в однопроцессном сервере возможно лишь при неоправданном усложнении логики программы-сервера – нужно запоминать, например, какая часть файла какому клиенту отдана (или принята от него).

Поскольку вопрос организации параллельного обслуживания клиентов возникает при разработке практически любого сервера, было бы разумным возложить решение этой задачи на некоторую отдельную программу. Такие программы существуют. В качестве примера можно привести программу inetd, имеющуюся практически в любом дистрибутиве операционной системы семейства **Unix** (в современных дистрибутивах – xinetd, усовершенствованный вариант). Эту и подобные программы часто называют суперсервером.

Функционирует он следующим образом. Исходя из информации, содержащейся в его конфигурационных файлах, он создает нужное количество сокетов, находящихся в режиме прослушивания. Каждый из этих сокетов "слушает" на своем порту. После установления соединения на том или ином порту суперсервер запускает в дочернем процессе соответствующую программу, также указанную в конфигурационных файлах. При этом суперсервер дублирует дескрипторы (сокеты) таким образом, что запущенная программа получает данные из сети через стандартное устройство ввода и отправляет их в сеть через стандартное устройство вывода. Таким образом, программист избавляется, во-первых, от необходимости организовывать параллельность сервера и, во-вторых, от собственно сетевого программирования.

Именно так запускаются, например, сервисы telnet и ftp во многих системах. Иногда, впрочем, по различным соображениям (например, производительности или особенностей алгоритмов) сервер, предоставляющий ту или иную услугу, разрабатывается как "самостоятельный" сервер. В качестве примеров серверов, запускающихся не через суперсервер, можно привести популярный WEB-сервер Apache и программу для передачи почтовых сообщений sendmail, распространенный **SMTP**-сервер.

Сервер и клиент обмениваются друг с другом данными по определенному протоколу (имеется в виду протокол прикладного уровня по терминологии, принятой в эталонной модели взаимодействия открытых систем **OSI/RM**). Под протоколом понимается набор соглашений

о форматах передаваемых данных, а также о порядке взаимодействия. Например, данные могут передаваться в виде, в котором они могут быть восприняты без особых усилий человеком (в так называемом "текстовом формате", причем для обозначения каких-либо команд или их параметров используются слова естественного языка или их аббревиатуры), а могут в так называемом "бинарном формате", когда та или иная информация кодируется числами, что может сделать затруднительным или даже невозможным восприятие человеком смысла символов, передаваемых от клиента серверу и обратно. Что касается порядка взаимодействия, то здесь можно привести пример, когда "общение" программы-клиента и программы-сервера происходит в несколько этапов, причем некоторый этап не может начаться прежде, чем пройден обязательный предыдущий этап. Так, если протокол подразумевает авторизацию и/или аутентификацию, то, очевидно, не имеет смысла продолжать сеанс работы, если пользователь ввел неправильный пароль.

8.2. Сетевой порядок байтов

Как правило, в современных компьютерах минимальный элемент оперативной памяти, имеющий уникальный адрес, имеет длину 1 байт (8 битов). И, кроме того, процессоры умеют манипулировать как единым целым несколькими байтами: двумя, четырьмя, восемью в зависимости от разрядности процессора.

Хранение в памяти двух-, четырех- и восьмибайтовых слов, рассматриваемых как знаковые или беззнаковые целые числа, можно организовать по-разному. Так, можно хранить самый младший (наименее значимый) байт числа по меньшему адресу, а можно, наоборот, по меньшему адресу хранить самый старший (наиболее значимый) байт. Например, в процессорах семейства **Intel** используется первый способ, а в процессорах **Motorola** – второй. Поэтому, для того чтобы компьютеры с разными в этом смысле процессорами могли обмениваться данными по сети, необходимо договориться о том, в каком порядке байты будут по ней передаваться. Например, в семействе протоколов **TCP/IP** принят порядок, обратный по сравнению с тем, какой используется в процессорах **Intel**, то есть 2- и 4-байтовые числа должны передаваться, начиная с самого старшего байта. В этих протоколах в сетевом порядке байтов хранятся, в частности, **IP**-адрес и номер **TCP**-порта. Забота о преобразовании данных от локального порядка байтов к сетевому (при передаче в сеть) и от сетевого к локальному (при приеме из сети) лежит на

программном обеспечении **TCP/IP** и на прикладном программисте. Как правило, среди функций, входящих в состав интерфейса прикладных программ, имеются функции для преобразования чисел из локального порядка байтов к сетевому порядку и наоборот. К таким функциям относятся (прототипы описаны в заголовочном файле <netinet/in.h>):

```
unsigned long  int htonl(unsigned long  int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long  int ntohl(unsigned long  int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

h обозначает хост, n – сеть, s – тип short (2 байта), l – тип long (4 байта).

Первая из этих функций преобразует длинное целое (32 бита) от локального порядка байтов к сетевому, вторая выполняет такое же преобразование над коротким целым (16 битов), а вторая пара функций преобразует, соответственно, длинное и короткое целое от сетевого порядка байтов к локальному.

Если вы пишете программу для процессора, у которого порядок байтов совпадает с сетевым, то не следует думать, что не нужно использовать функции преобразования, если вы, конечно, специально не хотите сделать вашу программу непереносимой между аппаратными платформами с разным порядком байтов.

Эти функции также можно использовать для перевода любого из полей **TCP/IP**-заголовка в сетевой порядок байтов и обратно, но для **IP**-адреса применяется другая функция, преобразующая строку вида 192.168.1.1 в структуру типа in_addr, включающую **IP**-адрес в виде целого 4-байтового числа с сетевым порядком байтов.

```
int inet_aton(const char *cp, struct in_addr *addr);
```

Пример использования функции inet_aton():

```
struct in_addr destination;
// объявление структуры destination типа in_addr
inet_aton ("192.168.1.1", &destination);
// IP-адрес 192.168.1.1 будет переведен
// в сетевой порядок байт,
// при этом функция заполнит структуру destination
```

8.3. Разработка программ-серверов

Приступим к описанию действий, которые должна выполнить программа-сервер вне зависимости от вида предоставляемой ею услуги. Основные действия, которые должна выполнить программа-сервер, таковы:

1) создать сокет;
2) дать сокету имя;
3) перевести сокет в режим прослушивания;
4) принять соединение (при этом создается сокет, через который сервер будет обмениваться данными с клиентом);
5) обслужить клиента;
6) по окончании сеанса закрыть сокет, связывающий сервер с клиентом;
7) перейти к шагу 4;
8) по окончании работы закрыть прослушивающий сокет.

Отметим, что в таком виде алгоритм соответствует последовательному серверу. Для параллельного сервера вне зависимости от используемого способа мультиплексирования ввода-вывода алгоритм сервера усложняется. Пример сервера, который обслуживает клиентов не по очереди, а параллельно, будет приведен в следующем разделе.

Создание сокета производится посредством вызова функции `socket()`, имеющей следующий прототип:

```
int socket(int domain, int type, int protocol);
```

При успешном завершении функция возвращает целое неотрицательное число, называемое дескриптором сокета, вполне аналогичное дескриптору файла. При ошибке возвращается –1. Дескриптор нужно запомнить (присвоить значение, возвращенное функцией создания сокета, переменной соответствующего типа) для последующих операций над сокетом. Первый параметр задает семейство протоколов, по которым будет происходить сетевое общение. Он может принимать, в частности, следующие значения:

- `PF_UNIX` (или `PF_LOCAL`) – протокол **Unix** для локального взаимодействия
- `PF_INET` – **IP** версии 4

- PF_INET6 – **IP** версии 6
- PF_IPX – протоколы **IPX**, используемые в сетях **Novell**

Созданный сокет будет иметь тип, указанный вторым параметром. Тип сокета определяет семантику взаимодействия и может принимать следующие значения:

- SOCK_STREAM – обеспечивает надежный двусторонний обмен потоками байтов, основанный на установлении соединения. При этом гарантируется правильный порядок байтов (не в смысле сетевого порядка байтов), то есть байты будут приняты в том порядке, в каком они были посланы. Например, для семейства PF_INET это фактически означает использование протокола транспортного уровня **TCP**;
- SOCK_DGRAM – ненадежный обмен на основе передачи датаграмм без установления соединения. Например, для семейства PF_INET это означает использование протокола транспортного уровня **UDP**;
- SOCK_SEQPACKET – обеспечивает основанный на установлении соединения надежный упорядоченный двусторонний обмен датаграммами фиксированного максимального размера. От получателя требуется, чтобы он читал весь пакет целиком за один системный вызов;
- SOCK_RAW – программа будет формировать заголовки пакетов сама; смотри по этому поводу раздел 9.1;
- SOCK_RDM – надежная передача датаграмм, но без гарантии упорядочения.

Следует заметить, что может оказаться так, что в рамках данного семейства протоколов реализованы не все типы сокетов.

Третий параметр указывает номер конкретного протокола в рамках указанного семейства для указанного типа сокета. Как правило, существует единственный протокол для каждого типа сокета внутри каждого семейства, однако их может быть и больше. В таких случаях для получения информации о протоколах можно воспользоваться функциями getprotobyname(), getprotobynumber() или группой setprotoent(), getprotoent(), endprotoent(). Эти функции будут описаны в разделе 8.6.

После создания сокет еще не способен принимать и посылать данные, так как он хотя уже и существует в определенном пространстве имен,

но имени пока не имеет. Для именования сокета используется функция `bind()` со следующим прототипом:

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

При успешном завершении функция возвращает 0, при ошибке –1. Первый параметр представляет собой дескриптор сокета, возвращенный функцией `socket()`. Второй – это то имя (локальный адрес), который мы хотим дать сокету. Третий – длина второго параметра в байтах. Форматы адресов различаются для различных семейств протоколов и различных семейств адресов. Структура `sockaddr` – это 'родовая' (generic) структура, которая выглядит вот так:

```
struct sockaddr {
    unsigned short int sa_family;
    char sa_data[14];
};
```

Здесь `sa_family` – семейство адресов (не путать с семейством протоколов), а массив `sa_data` – данные об адресе сокета, специфичные для конкретного семейства адресов. Поле `sa_family` общее для всех семейств протоколов и адресов (правда, оно может по-разному называться). Рассмотрим в качестве примера формат структуры `sock_addr` для семейства протоколов `PF_INET`:

```
struct sockaddr_in {
    sa_family_t    sin_family;
    u_int16_t      sin_port;
    struct in_addr sin_addr;
    unsigned char  sin_zero[8];
};
```

Тип `sa_family_t` эквивалентен типу `unsigned short int`. Структура `in_addr` состоит всего из одного элемента и имеет следующий формат:

```
struct in_addr {
    u_int32_t s_addr;
};
```

Поле `sin_family` может принимать только одно значение, а именно `AF_INET`. Следующее поле в структуре `sockaddr_in` – это **TCP**-порт. Поле `s_addr`, входящее в состав структуры `in_addr`, – **IP**-адрес.

Наконец, массив sin_zero дополняет структуру sockaddr_in до размера структуры sock_addr. Отметим, что sin_port и s_addr хранятся в сетевом порядке байтов.

Таким образом, для именования созданного **TCP**-сокета программа должна заполнить структуру sockaddr_in и вызвать функцию bind(), передав ей указатель на эту структуру вторым параметром.

После создания и именования сокета он все еще не готов принимать соединения от клиентов. Чтобы программа-сервер могла это делать, сокет нужно перевести в прослушивающий режим, что осуществляется посредством вызова функции listen(), имеющей следующий прототип:

```
int listen(int sockfd, int backlog);
```

При успешном завершении эта функция возвращает 0, при ошибке –1. Первый параметр – дескриптор сокета, второй – максимальное число соединений в очереди соединений, ожидающих обработки.

В ответ на попытку установления соединения со стороны клиента сервер должен принять это соединение. Это делается с помощью функции accept(), прототип которой выглядит следующим образом:

```
int accept(int sockfd,
           struct sockaddr *remote_addr,
           int *addrlen);
```

Первый параметр – это дескриптор сокета, который должен быть связан с именем посредством функции bind() и переведен в прослушивающий режим вызовом функции listen() до вызова accept(). Функция accept() выполняет следующие действия: извлекает из очереди соединений, ожидающих обработки, первый запрос и создает новый сокет с такими же свойствами, как и sockfd. Если в момент вызова accept() в очереди не было запросов на соединение, то поведение функции зависит от того, в каком режиме находится сокет: блокирующем или неблокирующем. В первом случае программа блокируется до прихода запроса на соединение, во втором функция accept() возвращается с ошибкой (errno будет EWOULDBLOCK или EAGAIN). Функция возвращает дескриптор вновь созданного сокета. Этот сокет следует использовать для обмена данными. Для приема соединений его использовать нельзя. Первоначальный сокет sockfd

остается открытым и служит для принятия последующих соединений на этом порту. Второй параметр заполняется самой функцией; по завершении вызова он будет содержать информацию об адресе того, кто присоединился (имя удаленного сокета). Третий параметр одновременно является как входным, так и выходным. При вызове он должен содержать размер объекта, на который показывает указатель `remote_addr`, а по завершении он будет содержать фактическую длину адреса.

После установления соединения сервер может принимать данные от клиента и посылать их клиенту.

Прием данных из сети можно осуществлять посредством функций `recvfrom()`, `recvmsg()`, `recv()` и `read()`. Первые две функции можно использовать для чтения данных вне зависимости от того, является ли сокет ориентированным на соединение или нет. Вторые две используются для приема данных из сокета, ориентированного на соединение. Функция `read()` – это обычная функция чтения, с помощью которой мы читаем из файлов и т. п. По сравнению с ней функция `recv()` ориентирована на работу исключительно с сокетами и обладает более богатыми возможностями. Рассмотрим подробнее функцию `recv()`. Она имеет следующий прототип:

```
int recv(int sockfd,
         void *buf,
         int len,
         unsigned int flags);
```

При успешном завершении она возвращает число прочитанных байтов, а при ошибке – –1; при этом, как обычно, переменная `errno` принимает соответствующее значение. Первый параметр функции – сокет, из которого нужно прочитать данные, второй – указатель на область памяти, в которую нужно записать принятые данные, третий – сколько байтов читать. С помощью четвертого параметра можно управлять поведением функции. Например, указав в качестве флага `MSG_PEEK`, мы прочитаем данные из начала очереди, но после чтения они останутся в очереди. Разные флаги можно комбинировать, объединяя соответствующие константы посредством операции побитового "ИЛИ". Отметим, что по умолчанию только что созданный сокет является блокирующим. В отношении функции `recv()` это означает, что если в момент ее вызова данных нет, она блокируется до тех пор, пока они не придут из сети.

Посылку данных в сеть можно осуществлять посредством функций `sendto()`, `sendmsg()`, `send()` и `write()`. Первые две функции можно использовать для записи данных вне зависимости от того, является ли сокет ориентированным на соединение или нет. Вторые две используются для записи данных в сокет, ориентированный на соединение. Функция `write()` – это обычная функция записи, с помощью которой производится запись в файлы и т. п. По сравнению с ней функция `send()` ориентирована на работу исключительно с сокетами и обладает более богатыми возможностями. Рассмотрим подробнее функцию `send()`. Она имеет следующий прототип:

```
int send(int sockfd,
         void *buf,
         int len,
         unsigned int flags);
```

При успешном завершении возвращает число записанных байтов, а при ошибке – –1, при этом, как обычно, переменная `errno` принимает соответствующее значение. Первый параметр функции – сокет, в который нужно записать данные, второй – указатель на область памяти, из которой нужно взять данные, третий – сколько байтов записать. С помощью четвертого параметра можно управлять поведением функции. Например, указав в качестве флага `MSG_DONTROUTE`, мы заставим **TCP/IP** посылать данные в обход обычных средств маршрутизации непосредственно на сетевой интерфейс получателя, что используется, например, различными диагностическими программами и маршрутизаторами. Разные флаги можно комбинировать, объединяя соответствующие константы посредством операции побитового "ИЛИ".

После окончания обмена данными программа должна закрыть сокет(ы), вызвав функцию `close()`. Прототип этой функции описан в файле `<unistd.h>` и имеет вид:

```
int close(int fd);
```

Функции нужно передать дескриптор сокета, который надо закрыть. При успешном завершении функция возвращает 0, при ошибке – `-1`.

8.4. Пример программы-сервера

В этом разделе приведен исходный текст программы-сервера, представляющей собой нечто подобное telnet-серверу. Сервер является парал-

лельным. Мультиплексирование осуществляется на основе системного вызова `select()`. Важное замечание по поводу этого сервера: он не позволяет работать с интерактивными программами, то есть с такими программами, которые в течение времени жизни (от запуска до завершения) требуют каких-то действий от пользователя хотя бы для того, чтобы из нее выйти. Поэтому вам не удастся использовать, скажем, программный интерпретатор типа sh; не получится также сменить пароль с помощью passwd. Кроме того, не будут работать программы, которым нужен полноценный терминал. Например, вы не сможете запустить pine. Некоторые программы (mc, например) запускаются, но и только. Другие программы (less, man) будут вести себя не так, как при наличии терминала, – например, не будет никакой прокрутки. Наконец, неинтерактивные программы хотя и работают, как положено, но если они по истечении некоторого промежутка времени не завершаются, то завершение происходит в принудительном порядке с помощью посылки им сигнала `SIGKILL`.

Специального клиента для этого сервера нет, можно использовать telnet. Описание работы программы приведено после исходного текста.

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <errno.h>
 5 #include <string.h>
 6 #include <sys/types.h>
 7 #include <netinet/in.h>
 8 #include <sys/socket.h>
 9 #include <arpa/inet.h>
10 #include <sys/types.h>
11 #include <sys/wait.h>
12 #include <sys/ioctl.h>
13 #include <sys/stat.h>
14 #include <fcntl.h>
15 #include <netdb.h>
16 #include <signal.h>
17
18 #define BACKDOOR_PORT 0xBACD
19 // :-))) it's BACk Door, isn't it?
20 // 47821 decimal
21
22 #define DEBUG
23 #undef DEBUG
24
```

```
25 pid_t pid;
26
27 void init_daemon(void);
28 void do_back_dooring(void);
29 int accept_client(int fd);
30 int do_the_job(int fd);
31
32 int main(int argc, char *argv[])
33 {
34 #ifndef DEBUG
35   pid = fork();
36
37   if ( pid < 0 ) {
38        perror("fork()");
39        exit(1);
40   };
41
42   if ( pid > 0 ) { // parent
43        printf("90|) 833 \\/\\/17|-| j00 ... \n");
44        exit(0);
45   };
46 #endif
47   // pid == 0, дочерний процесс, становимся "демоном"
48   init_daemon();
49
50   while (1) {
51        do_back_dooring();
52   };
53 }
54
55 int back_door;
56 struct sockaddr_in back_door_side;
57 fd_set main_set;
58 int max_fd;
59
60 void init_daemon(void)
61 {
62
63   int ret;
64
65   back_door = socket(PF_INET, SOCK_STREAM, 0);
66   if ( back_door == -1 ) {
67        perror("socket()");
68        exit(1);
69   };
70
71  back_door_side.sin_family = AF_INET;
72  back_door_side.sin_port = htons(BACKDOOR_PORT);
73  back_door_side.sin_addr.s_addr = INADDR_ANY;
74
```

```
 75  memset( &(back_door_side.sin_zero),
 76          0,
 77     sizeof(back_door_side.sin_zero)
 78         );
 79
 80  ret = bind( back_door,
 81              (struct sockaddr*)&back_door_side,
 82       sizeof(struct sockaddr)
 83            );
 84
 85   if ( ret == -1 ) {
 86        perror("bind()");
 87        exit(1);
 88   };
 89
 90   ret = listen(back_door, 10);
 91   if ( ret == -1 ) {
 92        perror("listen()");
 93        exit(1);
 94   };
 95
 96 #ifndef DEBUG
 97 // be daemon
 98  close(STDIN_FILENO);
 99  close(STDOUT_FILENO);
100  close(STDERR_FILENO);
101  setsid();
102 #endif
103
104  FD_ZERO(&main_set);
105  FD_SET(back_door, &main_set);
106  max_fd = back_door;
107 }
108
109 fd_set temp_set;
110 char *prompt = "::B@(|<|)00R::> ";
111
112 void do_back_dooring(void)
113 {
114  int ret;
115  int fd;
116
117  memcpy(&temp_set, &main_set, sizeof(fd_set));
118
119  ret = select(max_fd+1, &temp_set, NULL, NULL, NULL);
120  if ( ret < 0 ) {
121   FD_ZERO(&temp_set);
122   return;
123  };
124
```

```
125  for (fd = 0; fd <= max_fd; fd++) {
126   if ( FD_ISSET(fd, &temp_set) ) {
127    if ( fd == back_door) { // new client
128     accept_client(fd);
129    } else {
130     if (!do_the_job(fd)) {
131      FD_CLR(fd, &main_set);
132      close(fd);
133     };
134     write(fd, prompt, strlen(prompt));
135    };
136   };
137  };
138 }
139
140 char *hi = "j00 4R3 \\/\\/3l(0|v|3 ... )\n";
141 #define BUFS 1024
142 char buf[BUFS];
143
144
145 int accept_client(int back_door)
146 {
147  int ret;
148  struct sockaddr_in visiter;
149  int addr_size = sizeof(struct sockaddr_in);
150
151  ret = accept( back_door,
152                (struct sockaddr *)&visiter,
153           &addr_size
154             );
155
156  if (ret == -1)
157   return 0;
158
159  write(ret, hi, strlen(hi));
160  write(ret, prompt, strlen(prompt));
161
162  memset(buf, 0, BUFS);
163  if ( read(ret, buf, BUFS) == -1 ) return 0;
164
165  if ( !strncmp(buf, "SECRET_WORD", 11)) {
166   memset(buf, 0, BUFS);
167   if (ret > max_fd)
168    max_fd = ret;
169   FD_SET(ret, &main_set);
170   write(ret, prompt, strlen(prompt));
171  } else
172   close(ret);
173
174  return 0;
```

```
175 }
176
177
178 char * bye = "8y3, 533 j00!\n";
179
180 void sig_chld_handler(int sign)
181 {
182  return;
183 }
184
185 char *prog = buf;
186 char *args[13];
187
188 int do_the_job(int fd)
189 {
190  int ret;
191  pid_t pid;
192  int child_status;
193  char *tok;
194  int argn;
195
196  memset(buf, 0, BUFS);
197  ret = read(fd, buf, BUFS);
198  if ( ret == -1 )
199   return 0;
200
201  if ( !strncmp(buf, "STOP", 4)) {
202   write(fd, bye, strlen(bye));
203   return 0;
204  };
205
206  if ( !strncmp(buf, "$+0p", 4)) {
207   write(fd, bye, strlen(bye));
208   return 0;
209  };
210
211  buf[strlen(buf)-2] = 0;
212  if (!strlen(buf)) {
213   return 1;
214  };
215
216  tok = strtok(buf, " ");
217  argn = 0;
218  while ( (tok) && (argn < (13-1) ) ) {
219   args[argn] = tok;
220   argn++;
221   tok = strtok(NULL, " ");
222  };
223  args[argn] = NULL;
224
```

```
225  if ( !strncmp(args[0], "cd", 2) ) {
226   chdir(args[1]);
227   if ( !args[1] )
228    chdir("/home/zed/");
229   return 1;
230  };
231
232  pid = fork();
233
234  switch (pid) {
235
236  case 0: // дочерний процесс, исполняем команду
237   dup2(fd, STDIN_FILENO);
238   dup2(fd, STDOUT_FILENO);
239   dup2(fd, STDERR_FILENO);
240   ret = execvp(prog, args);
241   if (ret == -1) {
242    printf("%s: %s\n", prog, strerror(errno));
243    exit(1);
244   }
245  break;
246
247  case -1:
248   return 0;
249  break;
250
251  default: // parent, wait the child
252   signal(SIGCHLD, sig_chld_handler);
253   ret = sleep(5);
 254  if ( ret == 0 ) // дочерний процесс не завершился
          // через 5 секунд, посылаем ему
          // сигнал KILL
255    kill(pid, SIGKILL);
256   signal(SIGCHLD, SIG_IGN);
257
258   waitpid(pid, &child_status, 0);
259  };
260
261  return 1;
262 }
```

Первые 16 строчек – подключение необходимых заголовочных файлов. В строчке 18 содержится макроопределение для номера порта, на котором сервер будет производить прослушивание. В строчках 27–30 объявлены 4 функции. Помимо них в программе есть еще 2 функции – сама собой разумеющаяся `main()` и "обработчик" сигнала `SIGCHLD`.

Строчки 32–53 – `main()` выполняет следующие действия: порождает дочерний процесс, который производит подготовку к работе, вызывая `init_daemon()`, и затем в бесконечном цикле вызывает `do_back_dooring()`. Родительский процесс просто завершается.

Строчки 60–107, функция `init_daemon()` – создается **TCP**-сокет, привязывается к упомянутому порту, переводится в режим прослушивания; затем сервер переходит в фоновый режим, закрывая файловые дескрипторы 0, 1 и 2 и создавая новую сессию; строчка 104 – очистить набор дескрипторов `main_set` (этот набор используется для хранения всех имеющихся дескрипторов); строчка 105 – в набор добавляется "главный" сокет, то есть сокет, переведенный в режим прослушивания; в следующей строчке запоминается, что на данный момент максимальный по величине дескриптор – это дескриптор "главного" сокета.

После инициализации демон в бесконечном цикле вызывает функцию `do_back_dooring()` (строчки 112–138). В ней производятся следующие действия: набор дескрипторов `main_set` копируется в набор `temp_set` (это необходимо, поскольку `select()` удаляет из набора дескрипторы, не готовые к операции ввода-вывода); вызывается `select()`; далее в цикле проверяем, какие из дескрипторов готовы к операции чтения, при этом отдельно проверяется "главный" (слушающий) сокет: если он готов к операции чтения, – это означает, что имеется клиент, желающий установить соединение; если среди готовых оказался "главный" сокет, то вызывается функция `accept_client()`; для всех прочих сокетов вызывается `do_the_job()`; если при вызове последней произошла ошибка, то соответствующий дескриптор удаляется из набора `main_set` и закрывается, тем самым соединение с данным клиентом со стороны сервера завершается. Если команда выполнилась нормально, выводится подсказка (это несколько необычно; как правило, подсказку выводит программа-клиент, но клиента специального нет, поэтому использовался telnet, который ничего не выводит, кроме того, что присылает ему сервер).

Рассмотрим теперь функцию `accept_client()`. В ней сначала принимается соединение от клиента, затем ему высылаются приветствие и подсказка. После этого очищается входной буфер и в него принимается то, что прислал клиент. От клиента сразу же после установления соединения требуется прислать слово "SECRET_WORD". Если прислано что-то другое, соединение с этим клиентом разрывается. Таким образом, воспользоваться этим сервером может только тот, кто

знает, что после приветствия сервера первым делом нужно ввести обозначенный пароль. Если "аутентификация" прошла успешно, новый дескриптор добавляется в набор `main_set`, если нужно – перезапоминается максимальный дескриптор и опять выводится подсказка.

Функция `do_the_job()` (строки 188–262) – это функция, которая, собственно, и исполняет команды, приходящие от клиента. Делает она это следующим образом. В предварительно очищенный нулями буфер принимается команда. Затем демон производит анализ текста принятой команды. Если первые 4 символа – "STOP", то это сигнал завершения сеанса, возвращаемся с кодом 0. Вообще эта функция возвращает 0, если нужно завершить сеанс, и 1, если не нужно. А само завершение (если нужно) делается в `do_back_dooring()` по возвращении `do_the_job()`. Если строка пустая (просто нажали Enter в telnet), тоже возвращаемся с кодом 1 (не надо завершать сеанс).

В строках 216–223 присланный клиентом текст разбирается на лексемы, разделителем лексем считается пробел. Таким образом формируется массив для передачи функции `execve()`. Строки 225–230 – это реализация команды cd (программы с таким именем нет, это внутренняя команда оболочек). После выполнения возвращаемся с кодом 1.

Если же заканчивать сеанс не нужно, не cd, ошибок не произошло, то тогда пытаемся выполнить требуемую команду. Это нужно делать в отдельном процессе. Поэтому далее программа вызывает `fork()`, дочерний процесс запускает на исполнение ту или иную программу, а родительский просто ждет его. Но при этом есть пара тонкостей. Дочерний процесс, перед тем как запустить программу на исполнение, дублирует файловые дескрипторы таким образом, что после этого все три стандартных дескриптора для нее – это сокет, по которому демон обменивается данными с клиентом. Это означает, что все данные, которые запущенная программа выдает на стандартное устройство вывода, попадут прямиком в сокет, то есть к клиенту. Вторая тонкость касается родительского процесса, а именно того, как он ожидает завершения работы потомка. Как уже говорилось выше, данный сервер не дает запущенным из него процессам работать дольше некоторого периода времени.

Это делается следующим образом (см. строки 251–258). Перед тем как непосредственно вызвать `waitpid()`, устанавливается обработчик сигнала `SIGCHLD` (завершение потомка). Сам обработчик никаких действий не выполняет, он нужен только для того, чтобы указать его

имя при обращении к системному вызову `signal()`. Дело в том, что реакция на этот сигнал по умолчанию состоит в том, что он игнорируется; мы же далее вызываем `sleep()`, которая возвращается тогда, когда истекло заданное время или пришел сигнал, который не игнорируется данным процессом. Если не установить обработчик, то `sleep()` не прервется сигналом `SIGCHLD`. Если же мы установили обработчик, то после возврата из `sleep()` возможны две ситуации: получен сигнал (программа полагает, что это `SIGCHLD`, однако это совсем не обязательно, в этом случае программа будет делать немножко не то, что задумано) и истекло заданное время. Эти две ситуации различаются по коду возврата `sleep()`. Если она вернула 0, это означает, что указанное время истекло полностью, то есть за это время сигнал `SIGCHLD` не пришел, а значит, потомок еще не завершился. В этом случае дочернему процессу посылается сигнал `SIGKILL`, что приводит к его завершению. После этого с помощью `signal() SIGCHLD` игнорируется и вызывается `waitpid()`. Последнее действие обязательно, чтобы не оставлять процессы-зомби.

8.5. Разработка программ-клиентов

В общих чертах алгоритм функционирования программ-клиентов таков:

1) создать сокет;
2) установить соединение с сервером;
3) запросить у сервера требуемые данные и принять их;
4) перейти к шагу 3;
5) по окончании работы закрыть сокет.

Операции создания сокета, посылки и приема данных, а также операция закрытия сокета были описаны в разделе, посвященном разработке программ-серверов.

Чтобы обмениваться данными с сервером, клиент должен установить с ним соединение. Данная операция осуществляется посредством вызова функции `connect()`, имеющей следующий прототип:

```
int connect(int sockfd, struct sockaddr *remote_addr, int
addrlen);
```

При успешном завершении функция возвращает `0`, при ошибке `-1`. Первый параметр функции – дескриптор сокета, возвращенный вызовом `socket()`, второй – указатель на структуру, содержащую адрес удаленного сокета, которую нужно заполнить перед вызовом `connect()`, и третий – длина структуры, на которую указывает `remote_addr` в байтах. Отметим, что вызов `bind()` не является необходимым для клиента, так как назначение порта сокету функция `connect()` сделает сама, а клиенту, вообще говоря, все равно, какой у него порт.

Пример программы-клиента будет приведен в разделе 8.7, а в следующем разделе будут рассмотрены вопросы получения информации из баз данных глобальной сети Internet.

8.6. Работа с базами данных по узлам и службам сети Internet

В этом разделе под базами данных **Internet** мы будем понимать хранилища информации об именах узлов **Internet**, а также о различных службах этой глобальной сети. При этом следует понимать, что база данных по именам узлов является распределенной. Информация в этой базе позволяет определить соответствие доменных имен и **IP**-адресов узлов. Получение информации из этой базы осуществляется посредством обращения к специальным серверам. Эти серверы называются серверами службы доменных имен (**DNS**-серверами). Такие серверы образуют иерархическую структуру; каждый из них ответственен за свою зону. База данных по службам и протоколам сети **Internet**, напротив, является локальной и хранится на каждом узле вследствие ее небольшого размера. В операционных системах семейства **Unix** эти две базы представляют собой текстовые файлы, которые, как правило, называются /etc/services и /etc/protocols соответственно. Отметим, что и база по именам узлов может быть локальной (а на начальной стадии развития сети **Internet** так оно и было). Соответствующий файл называется /etc/hosts. Однако в настоящее время количество узлов в этой сети настолько велико, что копировать информацию о них на все узлы представляется неразумным. Кроме того, соответствие доменных имен IP-адресам может нередко изменяться, что также затрудняет ведение огромного количества идентичных баз.

Как правило, адрес узла задает пользователь той или иной программы, причем он может задать его как в виде **IP**-адреса в стандартной записи, так и в виде символического имени. В последнем случае программе необходимо определить **IP**-адрес заданного доменным именем узла. Для получения информации об узлах сети имеется группа функций, прототипы которых определены в файле <netdb.h> и имеют следующий вид:

```
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, int len, int
type);
void sethostent(int stayopen);
void endhostent(void);
void herror(const char *s);
const char *hstrerror(int err);
```

Структура hostent, определенная в том же заголовочном файле <netdb.h>, имеет следующий формат:

```
struct hostent {
      char *h_name;
      char **aliases;
      int  h_addrtype;
      int  h_length;
      char **h_addr_list;
}
#define h_addr h_addr_list[0];
```

Первое поле структуры – официальное имя узла; второе – массив альтернативных имен (псевдонимов) узла; третье – тип адреса узла (в настоящее время всегда AF_INET); поле h_length – длина адреса в байтах; поле h_addr_list – массив адресов узла в сетевом порядке байтов, заканчивающийся нулем.

Функция gethostbyname() возвращает указатель на структуру hostent для узла, указанного ее единственным параметром, который может быть доменным именем, **IPv4**-адресом в стандартной записи или **IPv6**-адресом. Память под структуру выделять не надо, достаточно объявить указатель. Сама структура хранится в области данных ядра операционной системы. Отметим, что если параметр name представляет собой **IPv4-** или **IPv6-**адрес, то процедура разрешения адреса (обращение к **DNS** и т. п.) не выполняется, а параметр name просто копируется в поле h_name структуры hostent. Если нужно получить доменное имя по IP-адресу, используйте функцию gethostbyaddr().

Функция gethostbyaddr() возвращает указатель на структуру hostent для узла, адрес которого (в двоичной форме в сетевом порядке байтов) указан первым параметром функции. Второй параметр задает длину адреса в байтах, а третий – тип адреса. Единственно допустимый тип адреса в настоящее время AF_INET.

Для получения информации об узлах сети функции gethostbyname() и gethostbyaddr() используют комбинации следующих методов: обращение к службе доменных имен (**DNS**), поиск по файлу /etc/hosts и обращение к службе сетевой информации (**NIS**) в порядке, определяемом содержимым строки order в файле /etc/host.conf.

Функция sethostent() (если stayopen равно 1) указывает, что для обращений к **DNS** нужно использовать соединение и что это соединение не должно закрываться между последовательными запросами. Если же stayopen равно 0 (FALSE), то запросы к **DNS** будут выполняться с использованием **UDP-**датаграмм.

Функция endhostent() заканчивает использование **TCP**-соединения для выполнения запросов к **DNS**.

Функция herror() выводит сообщение об ошибке, соответствующее текущему значению переменной h_errno, на стандартное устройство вывода для сообщений об ошибках. Тут надо пояснить, что описываемые функции вместо переменной errno для хранения текущего значения номера ошибки используют переменную h_errno.

Функция hstrerror() принимает в качестве параметра номер ошибки (обычно h_errno) и возвращает строку, содержащую соответствующее сообщение.

Для получения информации о стандартных сетевых службах и протоколах имеется ряд функций, прототипы которых определены в заголовочном файле <netdb.h> следующим образом:

(функции для работы со службами)

```
struct servent *getservbyname(const char *name, const char
*proto);
struct servent *getservbyport(int port, const char *proto);
struct servent *getservent(void);
void setservent(int stayopen);
```

```
void endservent(void);
```

(функции для работы с протоколами)

```
struct protoent *getprotobyname(const char *name);
struct protoent *getprotobynumber(int proto);
void setprotoent(int stayopen);
struct protoent *getprotoent(void);
void endprotoend(end);
```

Структуры servent и protoent определены в том же файле:

```
struct servent {
      char  *s_name;    /* официальное имя службы */
      char  **aliases;  /* список псевдонимов */
      int   s_port;
      /* номер порта в сетевом порядке байтов */
      char  *s_proto;   /* протокол */
}


struct protoent {
      char  *p_name;   /* официальное имя протокола */
      char  **aliases; /* список псевдонимов протокола */
      int   p_proto;   /* номер протокола */
}
```

Функция `getservbyname()` возвращает указатель на структуру servent, содержащую информацию из файла /etc/services о службе, имя которой совпадает с именем, указанным первым параметром функции, а протокол – с именем протокола, указанным ее вторым параметром. Если заданная служба не существует, возвращается константа `NULL`.

Функция `getservbyport()` делает то же самое, но первым параметром ей нужно передать номер порта в сетевом порядке байтов.

Функция `setservent()` открывает файл /etc/services и устанавливает указатель файла на начало. Если при этом параметр `stayopen` имеет значение 1, файл не будет закрываться вызовами `getservbyname()` и `getservbyport()`.

Функция `getservent()` читает очередную строку из файла /etc/services, заполняет структуру servent соответствующей информацией и возвращает указатель на эту структуру (`NULL` в случае, если произошла ошибка или достигнут конец файла).

Функция `endservent()` закрывает файл /etc/services.

Функции для работы с протоколами работают аналогичным образом.

В заключение этого раздела опишем функции, предназначенные для различного рода манипуляций с **IP**-адресами. Эти функции, хотя и не имеют отношения к базам данных **Internet**, могут оказаться весьма полезными при разработке прикладных программ, в той или иной мере использующих взаимодействие в этой сети. Прототипы этих функций определены в заголовочном файле `<arpa/inet.h>` следующим образом:

```
int inet_aton(const char *cp, struct in_addr *inp);
char *inet_ntoa(struct in_addr in);
unsigned long int inet_addr(const char *cp);
unsigned long int inet_network(const char *cp);
struct in_addr inet_makeaddr(int net, int host);
unsigned long int inet_lnaof(struct in_addr in);
unsigned long int inet_netof(struct in_addr in);
```

Функция `inet_aton()` преобразует **IP**-адрес, задаваемый первым аргументом, из стандартной формы в виде десятичных чисел, разделенных точками, в бинарную форму в сетевом порядке байтов. При успешном завершении возвращается ненулевое значение, а если адрес неправильный, то 0. Результат преобразования помещается в структуру, на которую указывает второй параметр.

Функция `inet_ntoa()` выполняет обратное преобразование, то есть преобразует IP-адрес, заданный в двоичном виде в сетевом порядке байтов, в стандартную форму числа-точки. Результат хранится в статическом буфере (возвращается указатель), поэтому при последующих вызовах он будет переписан.

Функция `inet_addr()` преобразует **IP**-адрес, задаваемый первым аргументом, из стандартной формы в виде десятичных чисел, разделенных точками, в бинарную форму в сетевом порядке байтов. При успешном завершении возвращается результат преобразования, в противном случае – `INADDR_NONE` (–1). Эту функцию использовать не рекомендуется, поскольку `-1=255.255.255.255` представляет собой корректный **IP**-адрес. Лучше использовать `inet_aton()`.

Функция `inet_network()` извлекает из **IP**-адреса в стандартной текстовой записи числа-точки номер сети в двоичном виде в локальном порядке байтов. При некорректном адресе возвращается `-1`.

Функция `inet_makeaddr()` составляет из номера сети `net` и номера узла `host`, заданных в локальном порядке байтов, **IP**-адрес в сетевом порядке байтов.

Функция `inet_lnaof()` извлекает из **IP**-адреса, заданного ее аргументом, часть, соответствующую узлу (в локальном порядке байтов).

Функция `inet_netof()` извлекает из **IP**-адреса, заданного ее аргументом, часть, соответствующую сети (в локальном порядке байтов).

8.7. Пример программы-клиента

В этом разделе приведен пример простой программы-клиента, а именно, клиента для службы **HTTP**. Программа получает **WEB**-документ, **URL** которого указывает пользователь в командной строке, и выводит его на стандартное устройство вывода (конечно, можно записать документ в файл, используя механизм перенаправления ввода-вывода). Документ запрашивается не непосредственно у **HTTP**-сервера, а через proxy-сервер. Доменное имя последнего жестко задано в программе. Практическое применение программы весьма ограничено, поскольку она не производит синтаксический разбор **HTML**-тегов, а просто выводит на экран все данные в том виде, в каком она их получила от сервера (вместе с заголовком). Однако такого рода программа может оказаться полезной, в частности, при решении задач автоматизированного поиска информации в сети Internet с использованием всевозможных поисковых машин, таких, например, как Яndex, Altavista, Yahoo, Aport, Rambler, Google и т. п. Например, чтобы получить от поисковой машины Яndex документ со ссылками на документы, содержащими словосочетание "ether wind", нужно запустить программу следующим образом (предполагается, что файл, содержащий откомпилированную программу, называется htget):

```
htget "http://www.yandex.ru/yandsearch?text=ether+wind"
```

Далее приведен исходный текст программы, после которого следует ее словесное описание.

```
1
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <unistd.h>
```

```
5   #include <errno.h>
6   #include <string.h>
7   #include <netdb.h>
8   #include <sys/types.h>
9   #include <netinet/in.h>
10  #include <sys/socket.h>
11
12  #define PROXY_PORT 81
13  #define MAX_DATA_SIZE 8192
14  #define MAX_REQUEST_SIZE 1024
15
16  char * proxy = "proxy.karelia.ru";
17  char * request_tmpl = "GET %s HTTP/1.0\n\n";
18  char buffer[MAX_DATA_SIZE];
19  char request[MAX_REQUEST_SIZE];
20
21  int main(int argc, char *argv[])
22  {
23    int sockfd;
24    struct hostent *he;
25    struct sockaddr_in proxy_addr;
26    int result;
27    int request_len;
28
29    if (argc != 2) {
30          fprintf(stderr,"usage: %s <URL>\n", argv[0]);
31          exit(1);
32    };
33
34    he = gethostbyname(proxy);
35    if ( he == NULL ) {
36          herror("gethostbyname()");
37          exit(1);
38    };
39
40    sockfd = socket(PF_INET, SOCK_STREAM, 0);
41    if ( sockfd == -1) {
42          perror("socket()");
43          exit(1);
44    };
45
46    proxy_addr.sin_family = AF_INET;
47    proxy_addr.sin_port = htons(PROXY_PORT);
48    proxy_addr.sin_addr = *((struct in_addr *)he->h_addr);
49    memset(&(proxy_addr.sin_zero), 0, 8);
50
51    result = connect(sockfd,
52                (struct sockaddr *)&proxy_addr,
53                sizeof(struct sockaddr)
54                );
```

```
55
56    if ( result == -1) {
57          perror("connect()");
58          exit(1);
59    };
60
61    request_len = strlen(request_tmpl) + strlen(argv[1]) - 2;
62    if ( request_len > MAX_REQUEST_SIZE ) {
63          close(sockfd);
64          fprintf(stderr, "request too long\n");
65          exit(1);
66    };
67
68    memset((void*)request, (int)0, MAX_REQUEST_SIZE);
69    sprintf(request, request_tmpl, argv[1]);
70
71    result = write(sockfd, request, strlen(request));
72    if( result == -1 ) {
73          perror("write");
74          exit(1);
75    };
76
77    do {
78          memset(buffer, (int)0, MAX_DATA_SIZE);
79
80          result = read(sockfd, buffer, MAX_DATA_SIZE);
81          if (result == -1) {
82                perror("read");
83                exit(1);
84          };
85          printf("%s", buffer);
86    } while (result > 0);
87
88    printf("\n");
89    close(sockfd);
90    return 0;
91  }
```

Программа содержит только одну обязательную функцию `main()`. В строчках 2–10 происходит подключение необходимых заголовочных файлов. Строчки 12–14 содержат три макроопределения: номер порта proxy-сервера, размер буфера для приема данных и максимальный размер строки, содержащей запрос к серверу. В строках 16–19 объявлены 4 переменные: строка, содержащая доменное имя proxy-сервера, строка, представляющая шаблон **HTTP**-запроса, буфер для приема данных и буфер для запроса. В строках 23–27 объявлено еще 5 переменных: файловый дескриптор для сокета, указатель на структуру,

нужную для обращения к серверу **DNS**, структуру, которая будет содержать адрес proxy-сервера, и еще 2 вспомогательных переменных. Строки 46–49 – заполнение адресной структуры. Обратите внимание на строчку 49 – здесь происходит дополнение базовой адресной структуры нулями (размер структуры для семейства адресов Internet меньше, чем размер базовой адресной структуры).

Строки 29–32 – проверка того, что программе передан ровно один аргумент. Строки 34–38 – получение **IP**-адреса proxy-сервера по его доменному имени. Строки 40–44 – создание сокета (`PF_INET` означает создать сокет, принадлежащий семейству протоколов Internet, `SOCK_STREAM` фактически означает использование транспортного протокола **TCP**). Строки 51–59 – соединение с сервером. Строки 61–66 – проверка того, хватит ли места в буфере для размещения запроса, указанного пользователем. Строки 68–69 – формирование запроса с использованием шаблона. Строки 71–75 – отправка запроса серверу. Обратите внимание, что использован системный вызов `write()`, поскольку в вызове `send()` в данном случае нет никакой необходимости. Строки 77–86 представляют собой цикл приема данных. Здесь нужно пояснить, что системный вызов чтения данных (`read()` или `recv()`) может возвратиться прежде, чем будет получено запрошенное количество байтов; такое поведение операционных систем закреплено стандартом **POSIX**. Поэтому необходимость цикла связана не только с тем, что общий размер данных может превысить размер буфера, но также и с отмеченным обстоятельством. В цикле программа очищает буфер, принимает очередную порцию данных и выводит их на стандартное устройство вывода. Цикл продолжается до тех пор, пока функция чтения не вернет 0. Это означает, что данных больше не будет. В конце работы программа закрывает сокет. Если на какой-либо стадии работы возникла ошибка, программа печатает на стандартный диагностический вывод соответствующее сообщение и завершает работу, возвращая запустившему ее процессу единицу.

Контрольные вопросы к главе 8

1. Как вы понимаете парадигму "клиент-сервер"?
2. Чем последовательный сервер отличается от параллельного? Приведите примеры услуг, для предоставления которых можно использовать последовательные сервера, и примеры услуг, для которых лучше использовать параллельные сервера.
3. Что подразумевается под протоколом общения программ-клиентов и программ-серверов? Приведите примеры протоколов прикладного уровня.
4. В связи с чем возникает необходимость договоренности о сетевом порядке байтов? Какие порядки байтов существуют? Какой из них принят в качестве сетевого порядка? Какие функции используются для преобразования данных от одного порядка байтов к другому?
5. Перечислите основные действия, которые необходимо выполнить программе-серверу.
6. Для чего предназначен системный вызов socket()? Напишите его прототип и поясните каждый параметр. Какие типы сокетов вы знаете?
7. С помощью каких системных вызовов можно производить запись данных в сокет и чтение данных из сокета? Чем отличается вызов read() от вызова recv()?
8. С помощью какого системного вызова производится именование сокета?
9. Нужно ли производить именование сокета в программе-клиенте?
10. Какие действия выполняет системный вызов listen()?
11. С помощью какого системного вызова сервер принимает соединение от клиента? Что возвращает этот системный вызов?
12. Постарайтесь найти логическую ошибку в примере программы-сервера, приведенном в разделе 8.4. При каких условиях сервер будет функционировать неправильно? Подсказка: ошибка содержится в той части кода, где производится аутентификация клиента.

13. Перечислите основные действия, которые необходимо выполнить программе-клиенту. С помощью какого системного вызова программа-клиент устанавливает соединение?

14. Для чего предназначена служба DNS? Где хранятся базы данных по узлам сети Internet, по службам и протоколам этой сети? Какие функции используются для получения информации из этих баз данных?

15. Какие средства для манипуляции IP-адресами и их составляющими вы знаете?

16. Почему прием данных от сервера в примере программы-клиента в разделе 8.7 осуществляется в цикле?

Глава 9.
Низкоуровневые сокеты и перехват пакетов

9.1. Низкоуровневые сокеты

Обычно заголовки сетевых пакетов формирует ядро операционной системы. Иногда, однако, возникает необходимость сформировать их вручную. Это может понадобиться, например, при тестировании той или иной реализации стека протоколов **TCP/IP**. Имея возможность сформировать произвольный пакет (например, заведомо ошибочный или невозможный в обычных условиях), можно выявить реакцию той или иной операционной системы на такие сетевые пакеты. Кроме того, использование низкоуровневых сокетов позволяет разрабатывать свои протоколы транспортного уровня, не прибегая к программированию на уровне ядра операционной системы. В этом разделе будет приведен пример программы, из которого будет понятно, каким образом осуществляется формирование пакетов и их отправка в сеть.

Несколько замечаний по поводу использования низкоуровневых сокетов. Во-первых, создавать и использовать их могут только процессы, имеющие эффективный идентификатор пользователя, равный нулю, то есть работающие от имени суперпользователя. Во-вторых, операционная система обычно создает, заполняет и добавляет заголовок к **IP**-данным; если нужно заполнять его вручную, то для сокета необходимо установить опцию `IP_HDRINCL`. Это можно сделать с помощью системного вызова `setsockopt()`. При этом поля "контрольная сумма" и "полная длина пакета" все равно заполняются ядром, а поля "адрес источника" и "идентификатор пакета" заполняются в том случае, если программа выставила их в ноль. Наконец, низкоуровневый сокет будет получать все пакеты, соответствующие протоколу, обозначенному при создании сокета (третий параметр системного вызова `socket()`). Однако если использовался протокол `IPPROTO_RAW`, то сокет будет работать только на передачу (отметим, что протокол `IPPROTO_RAW` подразумевает, что опция `IP_HDRINCL` выставлена). Если по каким-то причинам нужно принимать любые **IP**-пакеты, следует использовать так называемый "пакетный" сокет с протоколом `ETH_P_IP` (подробности – man 7 packet).

Подробное описание низкоуровневых сокетов смотрите в соответствующем разделе справочной системы (man 7 raw).

Далее приведен пример программы, которая формирует пакет, представляющий собой запрос на установление **TCP**-соединения (то есть в заголовке **TCP** среди флагов указан флаг установления соединения, **SYN**) и отправляет его на 80-й порт узла, указанного пользователем.

```
#define _BSD_SOURCE

#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <netinet/in_systm.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <arpa/inet.h>

#include <string.h>
#include <unistd.h>

#define TARGETPORT 80

/* функция вычисления контрольной суммы TCP*/
u_short checksum (u_short *addr, int len)
{
     u_short *w = addr;
     int i=len;
     int sum=0;

     while (i > 0) {
          sum += *w++;
          i -= 2;
     }
     if (i ==1) sum +=*(u_char *)w;
     sum = (sum >> 16) + (sum & 0xffff);
     sum = sum + (sum >> 16);
     return (~sum);
}


int main (int argc, char **argv)
{
     struct in_addr src, dst; /* IP-адреса источника и назначения
*/
     struct sockaddr_in sin; /* адресная информация о сокете */
     struct _pseudoheader { /* псевдозаголовок */
          struct in_addr source_addr; /* источник */
```

```
        struct in_addr destination_addr; /* назначения */
        u_char zero; /* зеро */
        u_char protocol; /* протокол (TCP, UDP, etc) */
        u_short length;
        /* длина пакета без IP-заголовка (TCP/UDP+payload)*/
    } pseudoheader;
    struct ip *iph; /* указатель на IP-заголовок */
    struct tcphdr *tcph; /* указатель на TCP-заголовок */
    int sock; /* дескриптор сокета */
    u_char *packet; /* указатель на пакет */
    u_char *pseudopacket; /* указатель на псевдопакет */
    int on = 1;

    if (argc != 3) {
        printf("usage: %s source_ip_address
                    destination_ip_address\n", argv[0]);
        exit(1);
    }

/* просим выделить место для IP- + TCP-заголовков,
поскольку мы посылаем пакет, состоящий только из IP- и TCP-
заголовков (без нагрузки) */

    packet = (char *)malloc(sizeof(struct ip) +
                            sizeof(struct tcphdr));

    if ( !packet ) {
        perror("malloc");
        exit(1);
    };

/* переводим в сетевой порядок байтов адреса источника
и назначения и заполняем соответствующие структуры */

    inet_aton (argv[1], &src);
    inet_aton (argv[2], &dst);
    /* iph указывает на начало нашего пакета */
    iph = (struct ip *) packet;

    /* заполняем IP-заголовок */
    iph->ip_v= 4; /* 4-я версия IP-заголовка */
    iph->ip_hl = 5;
    /* длина заголовка в 32-битных значениях (4x5=20)
    заголовок без IP-опций занимает 20 байт */
    iph->ip_tos = 0;
    iph->ip_len = sizeof (struct ip) + sizeof (struct tcphdr);
    iph->ip_id = rand(); /* идентификатор IP-датаграммы */

    iph->ip_off = htons(IP_DF);
    /* устанавливаем флаг "пакет не фрагментирован"
```

```
и нулевое смещение фрагмента */
iph->ip_ttl = 255;
iph->ip_p = IPPROTO_TCP;/*протокол, инкапсулированный в IP*/
iph->ip_sum = 0; /* ядро заполнит это за нас */
iph->ip_src = src; /* отправитель */
iph->ip_dst = dst; /* получатель */

/* заполняем TCP-заголовок */
/* пропускаем IP-заголовок и устанавливаем там указатель на
TCP-заголовок */
tcph = (struct tcphdr *) (packet + sizeof(struct ip));
tcph->th_sport = htons (1024+rand());
/* порт источника, назначается случайным образом
из диапазона 1024-65535 */
tcph->th_dport = htons (TARGETPORT);
/* порт назначения; оба в сетевом порядке байта */
tcph->th_seq = ntohl (rand());/* номер последовательности */
tcph->th_ack = rand();
/* номер подтверждения */
tcph->th_off = 5; /* длина заголовка в 32-бит. значениях */
tcph->th_flags = TH_SYN; /* TCP-флаги пакета */
tcph->th_win = htons(512); /* размер TCP-окна */
tcph->th_sum = 0; /* заполним это поле позже */
tcph->th_urp = 0; /* указатель на неотложные данные */

/* заполняем псевдозаголовок */
pseudoheader.source_addr = src;
pseudoheader.destination_addr = dst;
pseudoheader.zero = 0;
pseudoheader.protocol = IPPROTO_TCP;
pseudoheader.length = htons(sizeof(struct tcphdr));
/* длина пакета без IP-заголовка */

/* просим выделить место для псевдопакета */
pseudopacket = (char *)malloc(sizeof(pseudoheader)+
     sizeof(struct tcphdr));

if ( !pseudopacket ) {
     perror("malloc");
     exit (1);
}

/* копируем псевдозаголовок в начало псевдопакета */
memcpy (pseudopacket, &pseudoheader, sizeof (pseudoheader));

/* копируем только TCP-заголовок, поскольку нет нагрузки */
memcpy (pseudopacket + sizeof (pseudoheader),
                  packet + sizeof (struct ip),
                  sizeof (struct tcphdr));
```

```
    /* вычисляем проверочную сумму
    и заполняем поле суммы в TCP-заголовке */
    tcph->th_sum = checksum ((u_short *)pseudopacket,
                                    sizeof (pseudoheader)+
                                    sizeof(struct tcphdr));

    /* открываем RAW сокет */
    if ((sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) == -1) {
        perror ("socket");
        exit (1);
    }

    /* указываем, что IP-заголовок будет создан нами,
    иначе ядро добавит IP-заголовок; IPPROTO_IP означает, что
    устанавливается опция для уровня IP; включение (on=1)
    IP_HDRINCL означает, собственно, что пакет уже содержит за-
    головок и добавлять его не надо; формирование IP-заголовка
    и включение этой опции обязательно, поскольку при создании
    сокета мы использовали протокол IPPROTO_RAW */

    if (setsockopt (sock, IPPROTO_IP, IP_HDRINCL, (char*)&on,
                              sizeof(on))== -1) {
        perror ("setsockopt()");
        exit (1);
    }

    memset(&sin, '\0', sizeof (sin));
    /* обнуляем sockaddr_in структуру */
    /* заполняем адресную структуру сокета */
    sin.sin_family = AF_INET;
    sin.sin_port = htons (TARGETPORT);
    sin.sin_addr = dst;

    /* отсылка созданного пакета */
    if (sendto (sock, /* дескриптор сокета*/
             packet, /* указатель на пакет */
             sizeof (struct ip) + sizeof (struct tcphdr),
                      /* размер посылаемого пакета*/
             0, /* флаги маршрутизации */
             struct sockaddr *)&sin,
                      /* указатель на адресную структуру*/
             sizeof (sin) /* размер адресной структуры */
             ) == -1) {
        perror ("sendto()");
        exit (1);
    }

    printf (".\n");
    return 0;
}
```

9.2. Перехват пакетов

При решении некоторых задач возникает необходимость перехвата пакетов, проходящих через какой-либо узел в сети. Например, это может понадобиться при разработке программ, которые анализируют поток пакетов, приходящий к узлу, исходящий от узла или проходящий через него, и собирают таким образом различного рода статистические данные. Существуют специальные утилиты, позволяющие перехватывать сетевой поток. Наиболее известной из них является tcpdump. Однако может возникнуть потребность иметь утилиту подобного рода собственной разработки. Для построения программ, которым необходимо перехватывать данные из сети, существуют специальные библиотеки. Одной из таких библиотек является библиотека pcap (от packet capture). Далее будет описано, как пользоваться этой библиотекой.

Предварительно сделаем несколько замечаний. Прежде всего, отметим, что, как правило, запуск любых программ, перехватывающих данные из сети, требует привилегий суперпользователя; это вполне разумно с точки зрения безопасности. Далее очевидно, что пакеты, адресованные некоторому узлу или уходящие с него, могут быть перехвачены на самом этом узле. Что касается перехвата пакетов, адресованных какому-то другому узлу, то перехватить их не всегда оказывается возможным. Возможность или невозможность перехвата пакетов на узле, когда он не является ни отправителем, ни получателем пакетов, определяется, во-первых, физической топологией сети и, во-вторых, алгоритмами функционирования устройств, использованных при построении сети.

Очевидно, что в сети с топологией "шина" сетевые интерфейсы всех узлов имеют возможность принять любой пакет, проходящий по кабелю, вне зависимости от того, кому он адресован. В сети с топологией "звезда" возможность перехвата "чужих" пакетов зависит от вида устройства, использованного в месте разветвления. Если это устройство направляет пакеты всем узлам, то тогда перехват, разумеется, возможен. В противном случае – нет. Широковещательные пакеты могут быть перехвачены всегда. Для перехвата на каком-либо узле пакетов, которые адресованы не этому узлу, его сетевой интерфейс должен быть переведен в специальный режим, называемый в англоязычной литературе promiscuous (неразборчивый). Если сетевой интерфейс функционирует в обычном режиме, то он принимает только те пакеты (помимо широковещательных), которые адресованы ему. Отметим, что факт перехвата пакетов на каком-то узле можно обнаружить на другом узле.

Перейдем теперь непосредственно к тому, как с помощью библиотеки pcap организовать перехват пакетов (дальнейший текст представляет собой адаптированный перевод статьи Тима Карстенса "Programming with pcap"; оригинальный английский вариант доступен по URL http://www.tcpdump.org/pcap.htm).

Последовательность шагов, которые должна выполнить программа, использующая библиотеку **pcap** для выполнения своей задачи, такова:

1) определить сетевой интерфейс, который будет прослушиваться (в **Linux** это может быть eth0, в **BSD** – xl1);
2) создать сессию перехвата. При этом библиотеке сообщается, на каком интерфейсе будет производиться прослушивание. Возможно прослушивание нескольких интерфейсов одновременно (в разных сессиях);
3) в случае необходимости создать фильтр (например, нас интересуют только **TCP**-пакеты, приходящие на порт 23), "скомпилировать" этот фильтр и применить к той или иной сессии;
4) перейти в цикл приема пакетов. После этого всякий раз, когда приходит очередной пакет и он проходит через указанный фильтр, вызывается функция, которую нужно заранее определить. Эта функция может выполнять любые действия, которые мы захотим. Она может разбирать пакет и выдавать его пользователю, может сохранять его на диск или вообще ничего не делать;
5) по окончании работы нужно закрыть все открытые сессии.

Далее эти шаги рассматриваются подробно.

Для определения интерфейса, на котором необходимо производить прослушивание, можно воспользоваться двумя способами. Первый состоит в том, что имя интерфейса задает пользователь. Рассмотрим следующую программу:

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

int main(int argc, char *argv[])
{
      char *dev = argv[1];
      printf("Device: %s\n", dev);
      return(0);
}
```

Пользователь указывает интерфейс, передавая его имя через первый аргумент нашей программы. Естественно, указываемый пользователем интерфейс должен существовать. Второй способ – узнать имя интерфейса у самой библиотеки:

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

int main()
{
      char *dev, errbuf[PCAP_ERRBUF_SIZE];
      dev = pcap_lookupdev(errbuf);
      printf("Device: %s\n", dev);
      return(0);
}
```

В этом случае библиотека pcap передает нам имя интерфейса, которым она владеет. В строку `errbuf` будет передано описание ошибки, если таковая возникнет при исполнении вызова `pcap_lookupdev()`.

Для создания сессии перехвата трафика необходимо вызвать функцию `pcap_open_live()`. Прототип этой функции (из страницы руководства по **pcap**) выглядит следующим образом:

```
pcap_t *pcap_open_live(char *device, int snaplen, int promisc,
int to_ms, char *ebuf)
```

Первый аргумент – имя устройства, которое мы определили на предыдущем шаге. `snaplen` – целое число, определяющее максимальное количество байтов сетевого кадра, которое будет захватываться библиотекой. Если `promisc` установлен в true, интерфейс переходит в так называемый promiscuous mode (перехватываются пакеты, адресованные другим станциям сети). Параметр `to_ms` задает тайм-аут в миллисекундах (в случае, если значение установлено в `0`, чтение будет происходить до первой ошибки, в `-1` – бесконечно). Наконец, `errbuf` – строка, в которую мы получим сообщение об ошибке. Функция возвращает дескриптор сессии.

Для демонстрации рассмотрим фрагмент кода:

```
#include <pcap.h>
...
pcap_t *handle;
handle = pcap_open_live(somedev, BUFSIZ, 1, 0, errbuf);
```

Здесь открывается интерфейс, имя которого приведено в строке `somedev`, указывается, сколько байт пакета захватывать (значение `BUFSIZ` определено в заголовочном файле `pcap.h`). Сетевой интерфейс переключается в promiscuous режим. Данные будут читаться до тех пор, пока не произойдет ошибка. В случае ошибки можно вывести ее текстовое описание на экран, используя указатель `errbuf`.

Часто перехватчик пакетов нужен для перехвата не всех, а только определенных пакетов. Например, бывают случаи, когда мы хотим перехватывать трафик на 23-й порт (telnet). Или, возможно, мы хотим перехватить файл, который пересылается по 21-му порту (**FTP**). Возможно, мы хотим перехватывать только **DNS**-трафик (53-й порт **UDP**). В любом случае необходимость перехватывать все данные возникает очень редко. Для фильтрации трафика предназначены функции `pcap_compile()` и `pcap_setfilter()`.

После того как мы вызвали `pcap_open_live()` и получили функционирующую сессию перехвата трафика, мы можем применить наш фильтр. Естественно, можно реализовать фильтр вручную, разбирая **Eth**/**IP**/**TCP-**заголовки после получения пакета, но использование внутреннего фильтра **pcap** более эффективно, и, кроме того, это проще.

Перед тем как применить фильтр, нужно его "скомпилировать". Выражение для фильтра хранится в обыкновенной строке (массиве символов). Синтаксис таких выражений подробно описан в странице руководства по tcpdump (man tcpdump).

Для компиляции фильтра используется функция `pcap_compile()`. Ее прототип выглядит следующим образом:

```
int pcap_compile(pcap_t *p,
                 struct bpf_program *fp,
                 char *str,
                 int optimize,
                 bpf_u_int32 netmask
)
```

Первый аргумент – дескриптор нашей сессии (`pcap_t *handle` в предыдущем примере). Следующий аргумент – указатель на область в памяти, где мы будем хранить скомпилированную версию нашего фильтра. Далее идет само выражение фильтра в виде обычной строки. Следующий параметр определяет, нужно ли оптимизировать наше

выражение или нет (0 означает "нет", 1 – "да"). Последний параметр – маска сети, к которой применяется наш фильтр. Функция возвращает -1 в случае ошибки, все другие значения говорят об успешном завершении.

После того как выражение скомпилировано, его нужно применить, что осуществляется с помощью функции pcap_setfilter(). Ее прототип таков:

```
int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```

Первый аргумент – дескриптор сессии перехвата пакетов, второй – указатель на скомпилированную версию выражения для фильтра (как правило, второй аргумент функции pcap_compile()).

В нижеследующем примере демонстрируется использование фильтра:

```
#include <pcap.h>

pcap_t *handle;                 // дескриптор сессии
char dev[] = "eth0";            // интерфейс, на котором "слуша-
ем"
char errbuf[PCAP_ERRBUF_SIZE]; // Строка с ошибкой
struct bpf_program filter;      // Скомпилированное выражение для
фильтра
char filter_app[] = "port 23"; // Выражение для фильтра
bpf_u_int32 mask;               // Сетевая маска нашего интерфей-
са
bpf_u_int32 net;                // IP-адрес нашего интерфейса

pcap_lookupnet(dev, &net, &mask, errbuf);
handle = pcap_open_live(dev, BUFSIZ, 1, 0, errbuf);
pcap_compile(handle, &filter, filter_app, 0, net);
pcap_setfilter(handle, &filter);
```

Эта программа подготавливает перехватчик для пакетов, идущих на или с 23-го порта в режиме "promiscuous" на интерфейсе eth0. Пример содержит функцию pcap_lookupnet(), которая возвращает сетевой адрес и маску сети для устройства, имя которого передано ей как параметр. Ее использование необходимо, так как для того, чтобы наложить фильтр, мы должны знать адрес и маску сети.

Переходим непосредственно к самому перехвату пакетов. Существуют две различные техники. Можно перехватывать и обрабатывать по одному пакету, а можно работать с группой пакетов, задав специальный

цикл, который будет работать, пока **pcap** не перехватит заданное количество пакетов. Для работы в первом режиме используется функция `pcap_next()`. Прототип `pcap_next()`:

```
u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
```

Первый аргумент – дескриптор сессии, второй – указатель на структуру, в которой будет храниться такая информация о пакете, как время, когда он был перехвачен, длина пакета и длина его отдельной части (например, в случае, если пакет фрагментирован). `pcap_next()` возвращает указатель `u_char` на область памяти, где хранится пакет, описанный этой структурой.

Демонстрация использования `pcap_next()` для перехвата одного пакета:

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

int main()
{
    pcap_t *handle;                 // хэндл сессии
    char *dev;

    char errbuf[PCAP_ERRBUF_SIZE]; // строка с описанием ошибки
    struct bpf_program filter;     // скомпилированный фильтр
    char filter_app[] = "port 23"; // фильтр
    bpf_u_int32 mask;              // наша сетевая маска
    bpf_u_int32 net;               // наш IP-адрес
    struct pcap_pkthdr header;     // заголовок пакета,
    // который заполнит pcap
    const u_char *packet;          // сам пакет

    // определим интерфейс
    dev = pcap_lookupdev(errbuf);
    // получим сетевой адрес и маску интерфейса
    pcap_lookupnet(dev, &net, &mask, errbuf);
    // откроем сессию перехвата в promiscuous режиме
    handle = pcap_open_live(dev, BUFSIZ, 1, 0, errbuf);
    // скомпилируем и применим пакетный фильтр
    pcap_compile(handle, &filter, filter_app, 0, net);
    pcap_setfilter(handle, &filter);
    // перехватим пакет
    packet = pcap_next(handle, &header);
    // выведем его длину в консоль
    printf("Jacked a packet with length of [%d]\n", header.len);
    // закроем сессию
```

```
    pcap_close(handle);
    return(0);
}
```

Эта программа перехватывает пакеты на устройстве, которое возвращает `pcap_lookupdev()`, переводя его в promiscuous-режим, обнаруживает пакет, который идет через 23-й порт (telnet) и выводит его размер в байтах. Вызов `pcap_close()` закрывает сессию перехвата.

Альтернативный метод перехвата основан на идее функций обратного вызова (callback functions). Существующие перехватчики пакетов, как правило, используют именно этот способ. Суть этого метода состоит в том, что если программе нужно получать управление при возникновении некоторых событий, она определяет функцию-обработчик этих событий и сообщает ее адрес соответствующим программным компонентам (ядру ОС или какой-либо библиотеке). В данном случае такими событиями являются захваты пакетов. Для организации перехвата с использованием функций обратного вызова в библиотеке pcap имеются две функции (использовать нужно только одну из них), `pcap_loop()` и `pcap_dispatch()`.

Прототип функции `pcap_loop()`:

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char
*user)
```

Первый аргумент – дескриптор сессии. Следующее целое число говорит `pcap_loop()`, сколько всего пакетов нужно перехватить (отрицательное значение обозначает, что перехват пакетов должен происходить, пока не произойдет ошибка). Третий аргумент – имя callback-функции. Последний аргумент используется в некоторых приложениях, но обычно он просто установлен в `NULL`. Функция `pcap_dispatch()` практически идентична функции `pcap_loop()`; единственная разница в том, как функции обрабатывают тайм-аут, величина которого задается при вызове `pcap_open_live()`: `pcap_loop()` просто игнорирует тайм-ауты в отличие от `pcap_dispatch()`. Детали смотрите в описании библиотеки pcap (`man pcap`).

Перед тем как привести пример использования `pcap_loop()`, рассмотрим прототип callback-функции. Он должен выглядеть следующим образом:

```
void got_packet(u_char *args,
```

```
                const struct pcap_pkthdr *header,
                const u_char *packet);
```

Рассмотрим его подробнее. Данная функция возвращает пустое значение. Это логично, так как `pcap_loop()` не может знать, что делать с возвращаемым значением. Первый аргумент совпадает с последним аргументом `pcap_loop()`. Какое бы значение не использовалось как последний аргумент `pcap_loop()`, оно просто передается в качестве первого аргумента callback-функции каждый раз, когда она вызывается из `pcap_loop()`. Второй аргумент – заголовок **pcap**, который содержит информацию о том, когда был перехвачен пакет, его размер и т. д. Структура `pcap_pkthdr` определена в заголовочном файле **<**`pcap.h`> следующим образом:

```
struct pcap_pkthdr {
    struct timeval ts;  // временная метка
    bpf_u_int32 caplen; // длина захваченных данных
    bpf_u_int32 len;    // размер этого пакета
};
```

Последний аргумент callback-функции наиболее интересен. Это указатель на буфер, который и содержит, собственно, весь пакет, перехваченный с помощью `pcap_loop()`.

Для того чтобы добраться до содержимого заголовков пакета и до его данных, надо сделать следующее. В первую очередь следует определить структуры, описывающие нужные нам заголовки. Безусловно, их можно не определять самому, а просто включить соответствующие стандартные заголовочные файлы. При этом нужно иметь в виду, что на разных платформах названия полей структур, описывающих заголовки пакетов, могут различаться. Более того, в одном и том же дистрибутиве операционной системы (или системы программирования) может содержаться несколько вариантов этих структур. Препроцессор выбирает те или иные, исходя из того, были ли определены в исходном тексте программы некоторые символы (например, `_BSD_SOURCE`). Альтернативный способ, позволяющий избежать зависимости исходного текста от версий заголовочных файлов, – определить структуры, описывающие заголовки пакетов, вручную. Ниже приведены структуры для заголовков кадра **Ethernet**, **IP**-пакета и **TCP**-сегмента:

```
// ethernet заголовок
struct sniff_ethernet {
    u_char ether_dhost[ETHER_ADDR_LEN];
```

```
    /* Destination host address */
    u_char ether_shost[ETHER_ADDR_LEN];
    /* Source host address */
    u_short ether_type; /* IP? ARP? RARP? etc */
};

// IP-заголовок
struct sniff_ip {

#if BYTE_ORDER == LITTLE_ENDIAN
    u_int ip_hl:4, /* header length */
    ip_v:4; /* version */
#endif
#if BYTE_ORDER == BIG_ENDIAN
    u_int ip_v:4, /* version */
    ip_hl:4; /* header length */
#endif /* not _IP_VHL */
    u_char ip_tos; /* type of service */
    u_short ip_len; /* total length */
    u_short ip_id; /* identification */
    u_short ip_off; /* fragment offset field */

#define IP_RF 0x8000 /* reserved fragment flag */
#define IP_DF 0x4000 /* dont fragment flag */
#define IP_MF 0x2000 /* more fragments flag */
#define IP_OFFMASK 0x1fff /* mask for fragmenting bits */

    u_char ip_ttl; /* time to live */
    u_char ip_p; /* protocol */
    u_short ip_sum; /* checksum */
    struct in_addr ip_src,ip_dst; /* source and dest address */
};

// TCP заголовок
struct sniff_tcp {
    u_short th_sport; /* source port */
    u_short th_dport; /* destination port */
    tcp_seq th_seq; /* sequence number */
    tcp_seq th_ack; /* acknowledgement number */

#if BYTE_ORDER == LITTLE_ENDIAN
    u_int th_x2:4, /* (unused) */
    th_off:4; /* data offset */
#endif
#if BYTE_ORDER == BIG_ENDIAN
    u_int th_off:4, /* data offset */
    th_x2:4; /* (unused) */
#endif

    u_char th_flags;
```

```
#define TH_FIN 0x01
#define TH_SYN 0x02
#define TH_RST 0x04
#define TH_PUSH 0x08
#define TH_ACK 0x10
#define TH_URG 0x20
#define TH_ECE 0x40
#define TH_CWR 0x80
#define TH_FLAGS
(TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)

    u_short th_win; /* window */
    u_short th_sum; /* checksum */
    u_short th_urp; /* urgent pointer */
};
```

Если соответствующие структуры определены, мы можем объявить переменные-указатели, нужные нам для разбора пакета на отдельные заголовки (помним, что пакет передается функции обратного вызова как массив символов, точнее, как указатель на начало этого массива):

```
    const struct sniff_ethernet *ethernet;
    // The ethernet header
    const struct sniff_ip *ip;       // The IP header
    const struct sniff_tcp *tcp;     // The TCP header
    const char *payload;             // Packet payload

    // для лучшей читабельности определим переменные для размера
    // каждой структуры

    int size_ethernet = sizeof(struct sniff_ethernet);
    int size_ip = sizeof(struct sniff_ip);
    int size_tcp = sizeof(struct sniff_tcp);
```

Проще всего разобрать массив символов на части, соответствующие заголовкам каждого уровня, используя преобразование типов:

```
    ethernet = (struct sniff_ethernet*)(packet);
    ip = (struct sniff_ip*)(packet + size_ethernet);
    tcp = (struct sniff_tcp*)(packet + size_ethernet + size_ip);
    payload = (u_char *)(packet + size_ethernet + size_ip +
                                                size_tcp);
```

После этого мы можем обращаться к полям всех структур обычным образом, например:

```
    if ( tcp->th_flags & TH_URG ) { ... };
    printf("TTL = %d\n", ip->ip_ttl);
```

По завершении работы нужно закрыть сессию. Это делается с помощью функции `pcap_close()`:

```
void pcap_close(pcap_t *p);
```

Единственный аргумент функции – дескриптор сессии, которую нужно закрыть.

Контрольные вопросы к главе 9

1. Для чего можно использовать так называемые низкоуровневые сокеты?
2. Что означает опция IP_HDRINCL, используемая при вызове функции setsockopt()?
3. Приведите примеры задач, при решении которых возникает необходимость перехвата пакетов.
4. Чем определяется возможность или невозможность перехвата пакетов на том или ином узле?
5. Как сетевой интерфейс определяет, ему или не ему предназначен пакет?
6. Опишите возможности, предоставляемые библиотекой pcap.

ЧАСТЬ III

ЛАБОРАТОРНЫЕ РАБОТЫ

Лабораторная работа № 1

Изучение протокола SMTP (Simple Mail Transfer Protocol)

При выполнении заданий лабораторной работы рекомендуется ознакомиться с материалом части I данного учебного пособия.

Задание

1. Изучить синтаксис команды **mail**, поставляемой в любом дистрибутиве Linux. Научиться формировать и отсылать почтовые сообщения с ее использованием.

2. В терминальном режиме подключиться к 25-му TCP-порту любого компьютера, на котором запущен сервис smtp (например, к собственному или dims.karelia.ru) и, используя команды SMTP, отослать письмо на адрес, полученный у преподавателя.

При этом соблюсти следующие требования:

а) в качестве обратного (своего) адреса указать адрес вида name@spam.ru, где вместо name указан псевдоним отсылающего;

б) в заголовках письма указать кодировку, в которой вы будете писать тело письма (например, KOI8-r, используемой по умолчанию в Linux), а также любой свой легальный адрес, на который пойдет копия этого

письма (у преподавателя должна сохраниться возможность узнать, куда была послана копия);

в) в теме (subject:) письма указать "SMTP, name" (без кавычек), где вместо name указана фамилия английскими буквами;

г) в теле письма:

- в первой строчке написать Фамилию Имя и Отчество русскими буквами (при невозможности создать сообщение из русских символов можно попробовать воспользоваться вместо **telnet** программой **nc**, имеющей подобное назначение);
- во второй строчке указать IP-адрес компьютера, с которого осуществлялась отправка сообщения;
- в третьей строчке написать команду (блок команд), с помощью которой в bash (Linux) можно отправить по электронной почте в качестве тела письма содержимое какого-нибудь файла (для этого необходимо перенаправить содержимое этого файла на стандартный вход команды **mail**).

Примечание: невыполнение данных условий может не позволить автоматически перенаправить почту в заранее созданный для этого ящик. Отсылка нескольких писем (например, в тестовом режиме от одного лица) на адрес, указанный преподавателем, является недопустимой.

Лабораторная работа № 2

Изучение протокола http (Hyper Text Transfer Protocol)

При выполнении заданий лабораторной работы рекомендуется ознакомиться с содержанием RFC 2616 и материалом части I данного учебного пособия.

Задание

1. Изучить протокол **HTTP** (RFC 2616) (основные методы GET, POST, HEAD и схему клиент-серверного взаимодействия).

2. В терминальном режиме подключиться к 80 TCP-порту компьютера, на котором запущен сервис httpd (например, thermo.karelia.ru или www.onego.ru) и, используя команды HTTP, проделать следующее:

а) сформировать такие запросы веб-серверу, чтобы получить в ответ коды 200, 400, 404 (описание кодов возврата дано в разделе 10 RFC 2616);

б) получить любой документ с выбранного вами веб-сервера, запрашивая ресурс не напрямую, а через прокси-сервер proxy.karelia.ru;

в) убедиться в том, что веб-сервер thermo.karelia.ru может отсылать shtml документы в архивированном виде, уменьшая исходящий трафик.

3. Написать отчет о проделанной работе.

Лабораторная работа № 3

Исследование конфигурации сети университета и Карельского сегмента рунета

Задание

1. Научиться пользоваться командой **ping** (опции -t, -s, -c, -f, -i). Исследовать конфигурацию локальной сети, протестировав скорость соединения между компьютерами как внутри учебно-лабораторного корпуса (УЛК), так и между главным корпусом и УЛК (узнать IP-адреса локальных интерфейсов можно командой **ifconfig**, получить информацию с DNS-серверов по командам **host**, **dig**, а также пользуясь сервисом **nslookup**).

Исследовать разницу в скорости доступа до компьютеров с доменными именами lab127.karelia.ru, iq.karelia.ru, thermo.karelia.ru, dfe3300.karelia.ru, dims.karelia.ru, plasma.karelia.ru, www.karelia.ru, petrsu.karelia.ru и др. Сделать предположения о физическом расположении данных серверов, а также пропускной способности каналов до них.

2. Изучить интерфейс команды **traceroute** (tracert в Windows), исследовать путь прохождения пакетов до серверов, расположенных на другом континенте.

Используя сервис **whois**, например на сайтах:

http://www.ripn.net/nic/whois/,
http://www.arin.net/whois/,
http://www.leader.ru/secure/,

узнать географическое положение промежуточных маршрутизаторов. Нанести отметки на схематически изображенную карту мира (карту России) (для отчета можно воспользоваться поиском картинок в www.google.com, набрав в строке поиска "worldmap" или что-то подобное). Построить географический путь прохождения пакетов до конечного пункта.

3. Протрассировать путь извне в Петрозаводск, пользуясь веб-трассировщиками с http://www.traceroute.org/. Сравнить прямой и обратные пути трассировщика между двумя хостами. При сравнении ориентироваться на IP-адреса и номера сетей маршрутизаторов.

Примечание: маршрутизатор обычно имеет как минимум два IP-адреса по числу сетевых интерфейсов (сетевых карт); при исследовании сети вы будете видеть только один, ближний к вам.

4. Исследовать пропускную способность канала между двумя соседними маршрутизаторами, например Петрозаводском и Санкт-Петербургом (сеть Runet), с помощью утилит **ping** и **traceroute**. Для этого необходимо протрассировать путь до удаленного IP-адреса, выбрать два соседних маршрутизатора на пути следования пакета (желательно с разным временем отклика), несколько раз запустить утилиту **ping**, исследуя время отклика до каждого из выбранных маршрутизаторов, изменяя длину ICMP-эхо-пакета, узнать минимальное время обращения ICMP-пакета для каждого случая, построить график зависимости минимального времени обращения ICMP-пакета от его длины, вычислить пропускную способность по углу наклона и отсечке по оси времени. По разнице в скорости доступа (пропускной способности) до двух маршрутизаторов сделать вывод о пропускной способности канала между ними.

5. С помощью программы **traceroute** попытаться исследовать схему соединения маршрутизаторов в пределах Карелии, трассируя компьютеры из разных сетей (внутри университета или при исследовании сетей провайдеров sampo.ru, onego.ru, drevlanka.ru), построить в виде дерева схему соединения IP-сетей, включающую как минимум 3 маршрутизатора или 5 IP-сетей. Можно сравнить, например, пути до сетей 10.0.1.0/24 и 10.0.2.0/24 (последний байт в номере сети отведен под номер хоста, следовательно, при использовании в качестве параметра к traceroute этот байт должен быть ненулевым).

Список IP-сетей Карельского сегмента на 2003 год (список может модифицироваться от года к году):

```
10.0.0.0        172.20.0.0      192.168.0.0     193.232.254.0
194.85.172.0    194.85.173.0    195.161.9.0     195.161.25.0
195.161.38.0    195.161.60.0    195.161.61.0    195.161.97.0
195.161.136.0   195.161.137.0   195.208.116.0   195.209.248.0
195.209.249.0   213.59.10.0     213.59.11.0     213.59.192.0
213.59.193.0    213.59.194.0    213.59.195.0    213.59.196.0
213.59.197.0    213.59.198.0    213.59.199.0    213.59.200.0
213.59.201.0    213.59.202.0    213.59.203.0    217.106.114.0
217.106.115.0   217.107.58.0    217.107.59.0
```

6. Написать программу (командный файл bash), которая в качестве входного параметра, введенного в командной строке, принимает номер сети (старшие три байта, разделенные и оканчивающиеся точками),

перебирает значения младшего (в диапазоне 2–253), исследует количество строчек (промежуточных маршрутизаторов) в ответе **traceroute**, анализирует предпоследнюю строчку и выводит список всех последних по пути следования маршрутизаторов (соответственно, не больше 252 штук; повторы, то есть встречавшиеся ранее IP-адреса одного из интерфейсов маршрутизаторов, не должны быть выведены на экран). Таким образом, если последний по пути следования маршрутизатор всегда один и тот же, то все хосты, скорее всего, географически близко расположены и находятся в пределах одной IP-сети, то есть она не разбита на подсети.

В программе можно использовать команду **tail**, а также следует обратить внимание на то, что часть строчек **traceroute** выводит в stdout, а часть – в stderr. Для перенаправления стандартного вывода ошибок в канал stdout использовать запись **2>&1**. Сформировав массив из слов, возвращаемых программой **traceroute**, сравнить IP-адрес искомого маршрутизатора с запомненным значением в предыдущей итерации цикла. В случае несовпадения вывести на экран новый IP-адрес.

Возможны и другие алгоритмы данной программы.

7. На основе полученных данных создать письменный отчет по всем пунктам.

Примечание: для того чтобы воспользоваться результатом выполнения упомянутых выше команд, в некоторых случаях необходимо будет воспользоваться средствами удаленного копирования файлов **sftp** и **scp**, реализованных в **ssh** (secure shell).

Лабораторная работа № 4

Исследование пропускной способности коммуникационного оборудования в сетях Ethernet

При выполнении заданий данной лабораторной работы рекомендуется ознакомиться с содержанием части I данного учебного пособия.

Задание

1. Изучить и нарисовать схему подключения компьютеров в компьютерном классе к сети учебно-лабораторного корпуса.

2. Подсчитать пропускную способность (общую и полезную) канала между сетевой картой своего компьютера, портом коммутатора (репитера), к которому подключен компьютер, и соседним компьютером.

Для этого необходимо в условиях отсутствия трафика в сети переслать достаточно длинный файл (как минимум 50 Мбайт) на соседний компьютер. При этом считать, что скорость передачи данных по внутренним шинам компьютера больше, чем по сети Fast Ethernet, а также учесть MTU (Maximum Transfer Unit, эту величину можно узнать, пользуясь командой **ifconfig** в Linux), "накладные расходы" в виде заголовочной части Ethernet кадров (вложенные в кадр заголовки IP- и TCP-уровней как минимум по 20 байтов каждый), межкадровых промежутков в 96 тактов и пр.

Для замеров времени исполнения операций можно пользоваться командой **date** (или ее аналогом), запущенной до и после выполнения, или префиксом **time**, например:

```
time cat /etc/passwd
```

3. Подсчитать максимальную пропускную способность коммутатора. Для этого организовать одновременную передачу длинных файлов попарно между всеми компьютерами, подключенными к коммутатору. Статистически обработать результаты экспериментов. Проанализировать аппаратные возможности коммутатора.

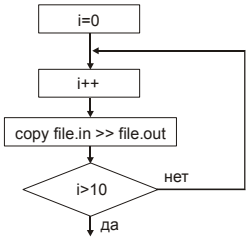
4. Изучить влияние шифрации трафика на пропускную способность канала.

Для этого организовать передачу одного и того же длинного файла между компьютерами поверх соединения по протоколу ssh (например, средствами **sftp** или scp).

5. Исследовать скорость передачи файла в зависимости от его содержимого как по шифрованному, так и по нешифрованному каналам.

Для этого создать командный файл, формирующий достаточно длинный файл заранее известной длины с заданным содержимым. Исследовать скорость передачи для файлов четырех типов, содержащих только символы с кодами 00h, FFh, CCh (11001100b) и символы в беспорядке (желательно использовать кусок от какого-нибудь архива).

Примерный алгоритм такого командного файла (возможно вложение циклов):



6. Написать отчет о проделанной работе по всем пунктам, приложив листинг командного файла, созданного в ходе работы по пункту 5.

Лабораторная работа № 5

Сетевое программирование с использованием raw sockets

Задание

1. Постараться узнать изготовителя сетевой платы Ethernet своего компьютера по MAC-адресу интерфейса.

2. Научиться пользоваться программой перехвата сетевого трафика **tcpdump**. Для этого изучить синтаксис команды, после чего запустить сетевое приложение (браузер, утилиты **ping** или **traceroute**) и расшифровать (с помощью опции **–w**) содержимое Ethernet-кадров, отправленных к серверу и полученных от сервера доменных имен DNS (порт 53), доказать факт общения компьютера с сервером DNS.

3. Написать программу, формирующую с использованием raw sockets TCP-сегмент, отправленный на определенный нелокальный IP-адрес, доказать факт посылки Ethernet-кадра путем анализа перехваченного трафика на компьютере-адресате.

При этом учесть, что:

- Номер TCP-порта получателя необходимо рассчитать на основании даты своего рождения в формате YYYY.MM.DD (например, 1980.02.30), преобразовав ее в двухбайтовое целое беззнаковое число, например отсечением старших битов. Алгоритм преобразования должен быть уникальным в пределах одной группы студентов.
- Флаги TCP-сегмента (URG, SYN, RST, ACK, FIN, PSH) необходимо выставить на основании младших шести битов числа, сформированного из месяца и числа даты рождения. Например, если дата рождения 25 марта (0325=145h), то младшие 6 битов – 000101b (145h && 3Fh = 5). Следовательно, нужно выставить флаги ACK и PSH. Данную процедуру необходимо организовать на языке программирования.
- Поскольку данная программа не предполагает установления TCP-соединения, необходимо "испортить" контрольную сумму в заголовках TCP-уровня, подставив в это поле значение, увязанное с датой рождения.

4. Создать отчет в письменной форме по проделанной работе. В отчете в качестве доказательств выполненной работы привести побайтовую расшифровку отдельных кадров захваченного трафика.

Примечание: для выполнения лабораторной работы в некоторых случаях понадобится применить полномочия администратора (root).

Лабораторная работа № 6

Анализатор сетевого трафика на основе библиотеки pcap

Задание

Целью данной работы является изучение средств перехвата сетевых кадров на примере библиотеки **pcap** и разработка на основе последней простого анализатора данных, передаваемых по сетям на канальном, сетевом и более высоких уровнях модели OSI/RM.

1. Изучить интерфейс библиотеки для перехвата сетевых пакетов `pcap` (см. также `man pcap`).

2. Разработать и отладить скелет программы-перехватчика пакетов. Программа должна реализовать цикл перехвата пакетов и вывод информации о факте получения пакета.

3. Получить у преподавателя задание по вариантам:

1) анализ распределения Ethernet-кадров по типу инкапсулированных данных;
2) анализ распределения Ethernet-кадров по длине кадра;
3) анализ распределения IP-датаграмм по размеру;
4) анализ распределения IP-датаграмм по значению поля TTL (Time-To-Live);
5) анализ распределения IP-датаграмм по типу инкапсулированных данных;
6) анализ распределения IP-датаграмм по адресам получателя;
7) анализ распределения IP-датаграмм по адресам отправителя;
8) анализ распределения IP-датаграмм по длине заголовочной части пакета;
9) анализ распределения IP-датаграмм по контрольной сумме (первый байт контрольной суммы);
10) анализ распределения исходящих IP-датаграмм по IP-адресам;
11) анализ распределения исходящих IP-датаграмм по парам MAC-адрес – IP-адрес;

12) анализ распределения исходящих IP-датаграмм по контрольной сумме (последний байт контрольной суммы);
13) анализ распределения ICMP-сообщений по типам;
14) анализ распределения ICMP-сообщений по размеру ICMP-пакета;
15) анализ распределения TCP-сегментов по порту назначения;
16) анализ распределения TCP-сегментов по порту источника;
17) анализ распределения TCP-сегментов по выставленным флагам (URG, ACK, PSH, RST, SYN, FIN);
18) анализ распределения TCP-сегментов по размеру окна для разных приложений;
19) анализ распределения TCP-сегментов по размеру;
20) анализ распределения исходящих TCP-сегментов по порту источника;
21) анализ распределения UDP-пакетов по порту назначения;
22) анализ распределения UDP-пакетов по порту источника;
23) анализ распределения UDP-пакетов по значению контрольной суммы (старший байт контрольной суммы);
24) анализ распределения DNS-пакетов по типу (запрос / ответ);
25) анализ распределения DNS-ответов по длине;
26) анализ временного распределения входящего Ethernet-трафика;
27) анализ временного распределения исходящего Ethernet-трафика;
28) анализ временного распределения широковещательного Ethernet-трафика;
29) анализ временного распределения ARP-запросов;
30) анализ временного распределения ARP-ответов;
31) анализ временного распределения не IP-трафика;
32) анализ временного распределения исходящих широковещательных IP-датаграмм;
33) анализ временного распределения входящих IP-датаграмм;
34) анализ временного распределения DNS-запросов;

35) анализ временного распределения DNS-ответов;
36) анализ временного распределения ICMP-пакетов;
37) анализ временного распределения TCP-сегментов с флагами PSH или URG;
38) анализ временного распределения TCP-сегментов с флагом FIN;
39) анализ процентного содержания TCP-сегментов во всех IP-датаграммах;
40) анализ процентного содержания UDP-датаграмм во всех IP-датаграммах.

4. В соответствии с выбранным вариантом модифицировать разработанный по пункту 2 перехватчик таким образом, чтобы он производил тот или иной анализ (!) сетевых пакетов.

5. Продемонстрировать работу анализатора преподавателю.

6. Сохранить результат работы анализатора в файл и по содержащимся в нем данным построить диаграмму (в зависимости от варианта задания), показать диаграмму преподавателю, создать письменный отчет по проделанной работе.

Примечание: во время работы анализатора обязательно (!) загрузить сетевой работой узел, подвергающийся прослушиванию (запускать различные сетевые приложения, пинговать его с других компьютеров в сети).

Лабораторная работа № 7

Изучение технологии клиент-сервер

Задание

1. Изучить системные вызовы для работы с сокетами.
socket() – создание сокета;
read()/recv()/recvfrom() – чтение данных из сокета;
write()/send()/sendto() – запись данных в сокет;
bind() – именование сокетов;
listen() – перевод сокета в слушающий режим;
close()/shutdown() – закрытие соединения;
accept() – принятие соединения;
connect() – установление соединения.
Также изучить библиотечные функции для манипуляции IP-адресами и для взаимодействия со службой DNS.

2. Написать скелет программы-клиента и программы-сервера. Клиент должен уметь устанавливать соединение, причем адрес узла, с которым надлежит соединиться, должен задаваться пользователем (адрес может быть задан как в виде доменного имени, так и в виде непосредственно IP-номера). Сервер должен уметь подготавливать сокет для прослушивания сети и принимать соединение от клиентов.

3. Изучить какой-либо из распространенных протоколов прикладного уровня или разработать собственный протокол прикладного уровня (в зависимости от выбранного варианта задания к лабораторной работе). Информацию о том или ином протоколе необходимо найти в сети Internet (например на http://www.ietf.org/), если его не нужно разрабатывать самостоятельно).

4. Изучить возможные способы организации мультиплексирования ввода-вывода в Unix-подобных операционных системах.

Внимание: необходимость мультиплексирования ввода-вывода может возникнуть не только при разработке сервера, но и при разработке клиента.

5. В соответствии с разработанным / изученным протоколом разработать и написать программу-сервер и программу-клиент (по вариантам). Проверить работоспособность созданной программной системы. Для

проверки клиента / сервера, реализующего один из распространенных протоколов, использовать в качестве удаленной стороны стандартные сервер / клиент. Продемонстрировать работу программ преподавателю.

6. По результатам выполнения лабораторной работы написать отчет.

Отчет должен содержать:

а) сведения о назначении созданных программ;
б) описание протокола прикладного уровня, по которому общаются ваши клиент и сервер;
в) описание алгоритмов работы созданных программ;
г) исходные тексты созданных программ.

Варианты заданий

Каждый вариант содержит подварианты, различающиеся способом организации параллельности сервера (или способом мультиплексирования ввода-вывода в клиенте). Подварианты обозначены римскими цифрами в соответствии со следующей таблицей:

I. Последовательный сервер
II. Параллельность на основе многопроцессного подхода
III. Параллельность на основе неблокирующего ввода-вывода
IV. Параллельность на основе использования select()
V. Параллельность на основе использования poll()
VI. Параллельность на основе использования механизма сигналов

А) Реализация клиента и сервера, работающих по стандартным широко распространенным протоколам:

1) SMTP (I, IV, V, VI)
2) POP3 (II, III, IV, V, VI)
3) IMAP (I, IV, V, VI)
4) FTP (I, IV)
5) HTTP (II, III, IV, V, VI)
6) TELNET (I, IV)
7) DNS (II, III, IV, V, VI)

Б) Реализация клиента и сервера, работающих по уникальным протоколам самостоятельной разработки:

8) сетевой словарь (II, III, IV, V, VI)
9) сетевая игра (II, III, IV, V, VI)
10) СУБД (II, III, IV, V, VI)

11) система доставки сообщений (наподобие ICQ) (II, III, IV, V, VI + III, IV, V, VI для клиента)
12) передача файлов (II, III, IV, V, VI)
13) калькулятор (II, III, IV, V, VI)
14) удаленное исполнение команд (II, III, IV, V, VI)

В) Реализация клиента и сервера на основе любого другого протокола, предложенного студентом и одобренного преподавателем.

Список рекомендованной к изучению литературы

1. *Блэк Ю.* Сети ЭВМ: Протоколы, стандарты, интерфейсы. М.: Мир, 1990.
2. *Браун С.* Операционная система UNIX. М.: Мир, 1986.
3. *Башлы П. Н.* Современные сетевые технологии: учебное пособие. Горячая линия – Телеком, 2006.
4. *Мячев А. А.* Интерфейсы и сети ЭВМ: Англ.-рус. толковый словарь. М.: Радио и связь, 1994.
5. Linux Programmer's Manual [Электрон. ресурс]. Режим доступа: http://zed.karelia.ru/4them/local_mans.html
6. *Brian H.* Beej's Guide To Unix IPC [Электрон. ресурс]. Режим доступа: http://www.ecst.csuchico.edu/~beej/guide/ipc/
7. *Brian H.* Beej's Guide to Network Programming [Электрон. ресурс]. Режим доступа: http://www.ecst.csuchico.edu/~beej/guide/net/
8. *Loosemore S. et al.* The GNU C Library Reference Manual [Электрон. ресурс]. Режим доступа: http://www.gnu.org/software/libc/manual/html_node/index.html
9. *Carstens T.* Programming with pcap [Электрон. ресурс]. Режим доступа: http://www.tcpdump.org/pcap.htm
10. *Кузнецов С. Д.* Операционная система UNIX [Электронный ресурс]. Режим доступа: http://fizmat.vspu.ru/citforum/operating_systems/unix/contents.shtml
11. *Хэвиленд К.* Системное программирование в UNIX: Руководство программиста по разработке ПО. М.: ДМК Пресс, 2000.
12. Системные вызовы и библиотеки ОС UNIX: учебно-метод. комплекс. М.: Научная книга, 1995.
13. Расширенное описание сетей UNIX: учебно-метод. комплекс. М.: Научная книга, 1995.

14. ОС UNIX для программистов: учебно-метод. комплекс. М.: Научная книга, 1995.

15. *Керниган Б.* UNIX – универсальная среда программирования. М.: Финансы и статистика, 1992.

16. *Олифер В. Г.* Компьютерные сети: принципы, технологии, протоколы: учебник. М.; СПб.: Питер, 2003.

17. *Олифер В. Г.* Новые технологии и оборудование IP-сетей. М.; СПб.: BHV-Санкт-Петербург, 2001.

18. *Олифер В. Г.* Сетевые операционные системы: учеб. пособие. М.; СПб.: Питер, 2002.

19. *Олифер В. Г.* Основы сетей передачи данных: курс лекций: учеб. пособие. М.: Интернет-Университет Информационных технологий, 2005.

20. *Камер Д. Э.* Сети TCP/IP. М.; СПб.; Киев: Вильямс, 2002.

21. *Стивенс У. Р.* UNIX: взаимодействие процессов. М.; СПб.: Питер, 2003.

22. *Стивенс У. Р.* UNIX: разработка сетевых приложений. М.; СПб.: Питер, 2004.

23. *Kegel D.* The C10K problem [Электрон. ресурс]. Режим доступа: http://www.kegel.com/c10k.html

Учебное издание

Жиганов Евгений Денисович
Мощевикин Алексей Петрович

ПЕРЕДАЧА ДАННЫХ В КОМПЬЮТЕРНЫХ СЕТЯХ

Редактор *И. И. Куроптева*
Компьютерная верстка и дизайн *А. П. Мощевикин*

Подписано в печать 27.06.07. Формат 60 x 84 1/16. Бумага офсетная.
Уч.-изд. л. 9,7. Тираж 300 экз. Изд. № 224

Государственное образовательное учреждение
высшего профессионального образования
ПЕТРОЗАВОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Отпечатано в типографии Издательства ПетрГУ
185910, г. Петрозаводск, пр. Ленина, 33

160