МФТИ

В. А. Мартиросян
В. В. Рубанов
Е. А. Шатохин

ОСНОВЫ ПРОГРАММИРОВАНИЯ В СРЕДЕ ОС Linux

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ)

ИНСТИТУТ СИСТЕМНОГО ПРОГРАММИРОВАНИЯ РАН

В. А. Мартиросян, В. В. Рубанов, Е. А. Шатохин

ОСНОВЫ ПРОГРАММИРОВАНИЯ В СРЕДЕ ОС LINUX

Допущено
Учебно-методическим объединением
высших учебных заведений Российской Федерации
по образованию в области прикладных математики и физики
в качестве учебного пособия для студентов вузов
по направлению «Прикладные математика и физика»

МОСКВА
МФТИ – ИСП РАН
2011

УДК 004.42(075)
ББК 32.973.2я73
М25

Рецензенты:

Кафедра системного программирования
Московского государственного университета им. М. В. Ломоносова
(зав. кафедрой академик *В. П. Иванников*)

Кандидат технических наук, доцент *С. М. Авдошин*

Мартиросян, В.А., Рубанов, В.В., Шатохин, Е.А.

М25 Основы программирования в среде ОС Linux**:** учеб. пособие / Мартиросян В.А., Рубанов В.В., Шатохин Е.А. – М.: ИСП РАН; МФТИ, 2011. – 116 с.

ISBN 978-5-7417-0348-8

Посвящено основам программирования в среде операционной системы Linux. Включает основные теоретические сведения и практические учебные примеры и задания, нацеленные на выработку базовых навыков программирования на уровне системных вызовов Linux, а также навыков разработки простых программ с графическим интерфейсом пользователя с использованием библиотек GTK и Qt.

Предназначено для студентов, аспирантов и научных сотрудников для обучения основным инструментам и навыкам программирования в среде ОС Linux.

УДК 004.42(075)
ББК 32.973.2я73

ISBN 978- 5-7417-0348-8

© Мартиросян В.А., Рубанов В.В., Шатохин Е.А., 2011

© Учреждение Российской академии наук «Институт системного программирования РАН», 2011

© Государственное образовательное учреждение высшего профессионального образования «Московский физико-технический институт (государственный университет)», 2011

Оглавление

Предисловие

Данное учебное пособие посвящено основам программирования в среде операционной системы GNU/Linux. Включает основные теоретические сведения и практические учебные примеры и задания, нацеленные на выработку базовых навыков программирования на уровне системных вызовов Linux, а также навыков разработки графических программ с использованием библиотек GTK и Qt. В качестве инструментальной основы используются популярные инструменты — компилятор GCC и среда разработки Geany.

Пособие предназначено для студентов, аспирантов и научных сотрудников, работающих в среде ОС Linux и желающих освоить основные инструменты программирования и особенности разработки программ в этой среде. Также в пособии затрагиваются общие методы программирования для Linux.

От читателя требуется знание языков программирования C и C++. Кроме общих навыков программирования от читателя требуется базовое знание операционной системы Linux на уровне пользователя (файловая система, командный интерпретатор и т.д.).

Пособие включает 4 главы. В первой главе рассматриваются особенности среды (окружения) работы программиста в ОС Linux и соответствующие методы и инструменты. Вторая глава содержит информацию об основных системных вызовах и системных примитивах Linux, доступных программистам приложений. Третья глава посвящена основам программирования графических пользовательских интерфейсов с помощью библиотеки GTK+. Четвертая глава дает введение в программирование графических приложений с использованием библиотек семейства Qt.

1. ПРОГРАММИРУЕМ В ОС LINUX: ОСНОВЫ

В этой главе описываются основные сведения, необходимые для разработки программ в среде операционной системы GNU/Linux (далее просто Linux), в том числе инструменты для ускорения, а иногда и автоматизации некоторых частей комплексного процесса разработки приложений.

Для тех, кто знаком с программированием в среде Windows, стоит отвыкнуть от тех средств, которые доступны разработчикам в этой среде, в частности от Microsoft Visual Studio. Также стоит понимать, что доступные системные библиотеки в Windows и Linux существенно отличаются.

1.1. Выбор среды разработки

В отличие от Windows, где разработчики преимущественно используют Microsoft Visual Studio или Borland/CodeGear C++ Builder, в Linux возможность выбора среды разработки приложений весьма широка. Множество альтернативных вариантов включает:

- текстовые редакторы без графического интерфейса (`emacs`, `vim`, `nano`, `mcedit`),
- текстовые редакторы с графическим интерфейсом (`Gedit`, `XEmacs`),
- специальные среды разработки приложений (`Geany`, `Kdevelop`, `Eclipse`, `Anjuta`, `Code::Blocks`, `QtCreator`).

Если в качестве среды выбран простой редактор, разработчику приходится вплотную иметь дело с компилятором и процессом компиляции (подробно об этом в 1.2). Тем не менее большинство редакторов обеспечивают такие возможности, как подсветку ключевых слов целевого языка программирования.

В данном пособии мы возьмем за основу среду разработки приложений — **Geany Lightweight IDE**, которая поддерживает следующие функции:

- подсветка ключевых слов, типов и т.д.;
- поддержка различных кодировок текста;
- автодополнение имен функций, переменных и типов;
- складывание текста (folding);

- встроенные команды компиляции, компоновки и запуска программ;
- возможность запуска `make` прямо из среды, в т.ч. с определением конкретной цели (make target) (см. 1.3);
- отладка программ;
- работа с системами контроля версий (SVN, CVS, GIT, ...).

Все вышеперечисленные функции делают Geany удобным инструментом для разработчиков программного обеспечения. Скачать Geany можно с официального сайта: http://www.geany.org. Для инсталляции программы нужно распаковать архив с исходными файлами, перейти в распакованный каталог, запустить скрипт `configure`. Если скрипт завершил работу без ошибок, то нужно запустить команду `make`, потом `make install`. Если все перечисленные шаги прошли успешно, то Geany можно будет запустить, как и все остальные программы (из командной строки или из основного меню).

Далее все примеры будут приведены с расчетом на Geany. Дополнительная справка о программе встроена в сам инструмент, а также доступна по адресу http://www.geany.org/Documentation/Manual.

1.2. Процесс компиляции и компоновки программы

В среде Linux компилятор и процесс компиляции являются важными понятиями не только для разработчиков, но часто и для пользователей. Поэтому мы постараемся детально ознакомить читателя с этими понятиями.

Основным компилятором в среде Linux является **GCC (GNU Compiler Collection)**. Это — большой проект, который включает в себя несколько компиляторов, в том числе компиляторы C, C++, Java, FORTRAN и другие. GCC поставляется со всеми известными дистрибутивами GNU. Более подробная информация о GCC и документация доступны на официальном сайте http://gcc.gnu.org.

Помимо GCC при разработке приложений для Linux нередко используются и другие компиляторы, например Intel® C++ Compiler. В данном пособии предполагается, что сборка приложения выполняется с помощью GCC версии не ниже 4.0.

Рассмотрим несколько сценариев компиляции проектов, что позволит в реальной обстановке изучить процесс компиляции. Начнем с самого простого случая: проект состоит из одного исходного файла,

написанного на C++ — `program.cpp`. В листинге 1.1 приведено содержимое файла.

Листинг 1.1: Hello World

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     cout << "Hello , world ! " << endl;
7     return 0;
8 }
```

Для его компиляции следует запустить следующую команду:

```
g++ program.cpp -o program
```

Эта команда запускает компилятор `g++`, передав в качестве первого параметра имя исходного файла. Второй параметр задает имя выходного файла. Следует заметить, что по умолчанию (без параметра `-o`) `g++` на выходе создает исполняемый файл с именем `a.out`, что не всегда удобно.

Для того чтобы перейти к более общему случаю, нужно ознакомиться с этапами компиляции. Конечно, то, о чем будет идти речь далее, свойственно не только GCC. Через эти этапы проходит любая программа на языке высокого уровня в процессе компиляции.

Этап	**Результат**
препроцессор (preprocessor)	исходный код без директив препроцессора (#)
компилятор (compiler)	исходный код на ассемблере
ассемблер (assembler)	объектный код
компоновщик (linker)	исполняемый код

На рисунке 1.1 показана схема разработки программ в Linux с помощью инструментов GCC (в том числе показано, что в программу можно включать модули, написанные непосредственно на ассемблере).

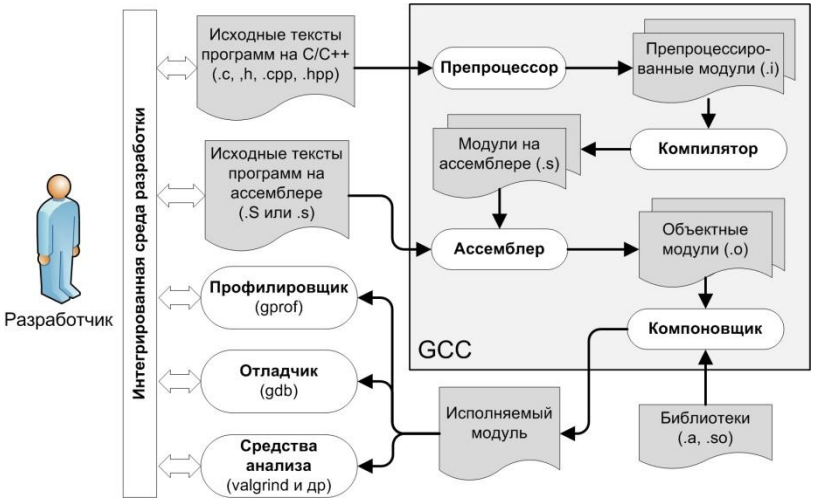

Рис. 1.1. Общая схема разработки программ с помощью GCC

Примечание 1.2.1. GCC на самом деле представляет собой интегрирующую оболочку, которая по умолчанию скрывает от пользователя (автоматизирует) трансформацию программы от исходных кодов до исполняемого модуля, выступая в роли «компилятора в широком смысле». Поэтому пользователи GCC по умолчанию не видят всех этих шагов и промежуточных файлов. Однако профессиональным программистам следует понимать представленную на рисунке внутреннюю схему работы.

GCC позволяет завершить процесс компиляции на любой из стадий. Для этого, соответственно, используются опции `-E`, `-S`, `-c`. Для наглядности рекомендуется самостоятельно опробовать приведенные опции и посмотреть результаты работы компилятора.

Итак, рассмотрим проект, который состоит из трех файлов: `main.cpp`, `print.hpp` и `print.cpp`. В листингах 1.2, 1.3 и 1.4 приведено содержимое этих файлов.

Листинг 1.2: main.cpp

```
1 #include ”print.hpp”
2
3 int main()
4 {
5     print();
6     return 0;
7 }
```

Листинг 1.3: print.hpp

```
1 #include <iostream>
2 using namespace std;
3
4 void print();
```

Листинг 1.4: print.cpp

```
1 #include ” print.hpp”
2
3 void print()
4 {
5     cout << ”Hello, world!” << endl;
6 }
```

Для сборки такого проекта нужно отдельно скомпилировать исходные файлы `main.cpp` и `print.cpp`, а потом скомпоновать полученные объектные файлы. Посмотрим, как это можно сделать при помощи GCC:

```
g++ –c print.cpp
g++ –o main print.o main.cpp
```

Первая команда компилирует исходный файл `print.cpp` и создает объектный файл `print.o` (этот параметр не задан, так как объектные файлы GCC по умолчанию именует именно таким образом). Вторая команда носит составной характер. Во-первых, она компилирует файл `main.cpp`, и только после этого компоновщик пытается найти недостающие символы (в данном случае в файле `main.cpp` не хватает функции `print()`). Для этого компоновщик сканирует все остальные файлы, которые переданы в качестве входных параметров (в нашем случае `print.o`). Это не единственный случай, когда какие-то функции или переменные находятся вне основного файла. Более

того, в большинстве случаев разработчикам приходится использовать какие-то библиотеки. Ярким примером служит библиотека `pthread` (POSIX Threads). Для того чтобы собрать программу, которая использует эту библиотеку, нужно каким-то образом дать компилятору знать о местоположении и имени файла, который содержит реализацию используемых функций. Такие библиотечные файлы бывают двух типов: *динамически* и *статически* компонуемые. Первый тип также называют *разделяемым объектом* (shared object), такие файлы имеют расширение "`.so`". Эти файлы являются аналогами динамических библиотек DLL в среде Windows. Они динамически подключаются к программе во время выполнения. Другие же файлы называют библиотечными архивами ("library archive"). Они представляют собой коллекцию объектных файлов, которые можно скомпоновать с вашими исходными файлами статически и не зависеть от каких-либо файлов во время выполнения программы. Оба этих подхода имеют свои положительные и отрицательные стороны. В первом случае недостатком является зависимость от внешних библиотечных файлов (.so файлов), во втором же случае - это большой размер программы и невозможность обновлять используемые библиотеки без пересборки приложения. Положительными сторонами являются соответственно небольшой размер программы и независимость от библиотек. Эти обстоятельства толкают создателей библиотек на создание двух версий библиотек: динамически и статически компонуемых. Более подробно о создании и использовании библиотек см. 1.8.

Кроме вышеуказанных, GCC обладает еще рядом возможностей. В частности, есть опции, контролирующие уровень педантичности к предупреждениям (`-Wall, -pedantic`) или управляющие внедрением отладочной информации в исполняемый файл (подробно об отладке см. 1.4). Более подробно о GCC можно прочесть в руководстве, запустив команду `man gcc` или `man g++`.

1.3. Инструмент GNU Make

Как говорилось выше, в среде Linux существуют инструменты, которые помогают разработчикам в процессе создания программного обеспечения. Одним из таких инструментов является утилита `GNU make` (http://www.gnu.org/software/make). Для начала обсудим вопрос необходимости такой утилиты. Вернемся к процессу компиляции проекта из нескольких файлов. Для получения результата необходимо

сделать несколько шагов (запустить несколько команд). Конечно, эти команды можно запускать вручную, как и делалось в примере. Но как быть, если количество файлов велико (например, сотни)? Есть альтернативный способ — создать скрипт (shell script), в котором поочередно будут прописаны команды компиляции. Но такой подход тоже имеет свои недостатки. Допустим, после успешной компиляции 50-ти файлов в следующем файле обнаружилась ошибка. Очевидно, что весь процесс должен быть приостановлен. После исправления ошибки, запустив скрипт, придется снова пройти через процесс компиляции всех предыдущих уже скомпилированных файлов. Разумеется, когда проект не слишком велик, то это не очень большая проблема. Но иногда проекты могут содержать тысячи файлов, перекомпиляция которых может занимать значительное время.

Решение этой проблемы предоставляет именно утилита `GNU make`. Make — это язык и его интерпретатор. Язык `make` является очень простым. В нем выделяется 3 ключевых понятия: *цели* (targets), *зависимости* (dependencies) и *команды* (commands). Ниже приведен пример скрипта на языке `make` (обычно эти файлы именуются `makefile`).

```
all : subtarget1 subtarget2
            command1
            command2
     subtarget1 : file1
            command3
     subtarget2 : file2
            command4
```

В этом примере описывается цель `all`, которая зависит от целей `subtarget1` и `subtarget2`, и для ее достижения нужно выполнить команды `command1` и `command2`.

Примечание 1.3.1. Обратите внимание, что каждая строка, содержащая команду, должна начинаться с двух символов табуляции.

Далее, рекурсивно цель `subtarget1` зависит от цели `file1`, и для ее достижения нужно выполнить команду `command3`. Что касается `file1`, то это уже имя файла, и как цель она уже готова для использования. То есть рекурсия тут прекращается. При повторном запуске `make` для всех целей сверяются даты модификации цели и ее зависимостей. Если ни одна из зависимостей не обновлялась после послед-

него запуска make, то эту цель не нужно повторно обрабатывать. Это приводит к тому, что если вы отредактировали один исходный файл, то повторно обработаны будут только те цели, которые зависят от этого файла.

В листинге 1.5 приведено содержимое makefile для рассмотренного выше примера.

Листинг 1.5: makefile

```
 1
 2 all : main.cpp print.o
 3       g++ -g -o main print.o main.cpp
 4
 5 print.o : print.cpp print.hpp
 6        g++ -g -c print.cpp
 7
 8
 9 clean :
10      rm *.o main
11
12 .PHONY: all clean
```

Следует отметить несколько специальных целей. Цель clean обычно вставляется во все скрипты make и предназначена для удаления всех выходных файлов. Также часто определяются цели install и uninstall соответственно для инсталляции и деинсталляции продукта. Приведенная в листинге цель .PHONY имеет системное значение. Она определяет, что цели all и clean должны быть обработаны в независимости от того, существуют ли файлы с такими же именами или нет.

Приведенный выше пример makefile иллюстрирует лишь узкий сектор возможностей make. В частности, в скриптах make можно определять переменные, которые также могут быть переопределены извне. Еще можно включать другие скрипты и возможно рекурсивно обработать составные проекты по всем каталогам. Более подробно об утилите make можно прочесть в руководстве, запустив команду man make.

Кроме утилиты make, существует еще ряд утилит, которые частично автоматизируют процесс разработки приложений. Среди таких систем достаточно широко используются autotools (Autoconf, Automake, Libtool, Gettext), CMake, Ant, Waf, SCons и другие.

1.4. Отладка

Процесс отладки, как и сам процесс разработки приложений, может быть реализован при помощи различных сред и утилит. В среде Linux основным является инструмент `gdb`. Существует плагин для отладки в Geany с помощью `gdb`. Он не входит в пакет Geany, но скачать его можно с официального сайта: http://plugins.geany.org/geanygdb/. Для активации нужно перезапустить Geany, зайти в меню *Tools->Plugin Manager*, найти *Debugger* и активизировать. После этого с левой стороны главного окна Geany появится новая вкладка "*Debug*". В ней можно загрузить исполняемый файл ("*Load*"), запустить его ("*Run*") и так далее. Однако отладить можно только те исполняемые файлы, в которых внедрена отладочная информация.

Внедрение отладочной информации реализуется посредством передачи соответствующих параметров компилятору. Следующая команда генерирует исполняемый файл с отладочной информацией (обратите внимание на параметр `-g`):

```
g++ -g sample.cpp -o sample
```

Полученный в результате файл можно загрузить в отладчик и запустить. Но прежде следует поставить точки остановки (`breakpoints`), что можно сделать при помощи кнопки "*Breaks*". Также можно поставить пусковой механизм (trigger) на запись или чтение каких-либо переменных (кнопка "Watches"). Управление пошаговым исполнением программы можно осуществлять, используя кнопки "*Next*", "*Next in*", "*Step*", "*Step in*", "*Return*".

Более подробную информацию о процессе отладки можно найти в руководстве `gdb` (`man gdb`). Плагин Geany является всего лишь визуальной оболочкой для `gdb`.

1.5. Поиск справочной информации

Как можно было заметить, почти во всех параграфах для нахождения более подробной информации мы ссылались на руководства пользователя соответствующих программ или утилит. Это делалось специально, чтобы подтолкнуть читателя к освоению утилиты `man`. Это один из основных источников информации в среде Linux. Почти все программы, которые инсталлируются в системе, имеют свои руководства, которые копируются в соответствующие каталоги для того,

чтобы они стали доступными из утилиты man. Основной каталог расположения руководств пользователя — это /usr/share/man. Руководства распределяются в подкаталоги в соответствии с секциями man. Предопределено восемь секций, основными из которых являются следующие:

(1) пользовательские команды,
(2) системные вызовы,
(3) функции стандартных библиотек,
(8) команды системного администрирования.

Иногда разные программы или функции могут иметь одно и то же имя. Для точной идентификации можно отмечать нужную секцию руководства следующим образом:

```
man 3 printf
```

Эта команда выдаст описание функции printf из стандартной библиотеки C, в то время как команда man printf по умолчанию выдаст описание системной утилиты printf.

Обычно руководства пользователя не содержат всю необходимую информацию. Для таких случаев можно воспользоваться еще одним источником информации — утилитой info.

Кроме указанных выше источников, разработчики программного обеспечения могут использовать еще один важный источник информации — исходные коды и заголовочные файлы. Самую достоверную информацию о какой-либо программе можно найти напрямую из ее исходных файлов ее реализации. Краткую информацию о программе или библиотеке можно найти из заголовочных файлов, которые обычно расположены в каталоге /usr/include.

1.6. Взаимодействие со средой

Каждая программа выполняется в определённой среде. В качестве среды можно рассматривать саму операционную систему или же множество данных, которые передаются от родительского процесса. И поскольку есть среда, то программа должна иметь возможность взаимодействовать с этой средой — получать и передавать различные данные.

Самый очевидный способ взаимодействия со средой — это передача аргументов. Как известно, все значения, которые прописаны по-

сле имени программы, передаются как аргументы функции `main()`. Таким образом, во время запуска программы ей можно передать любое количество строковых значений (если требуется передать численные значения, то программа будет вынуждена конвертировать переданные строковые значения в цифры). Обратная связь между процессом и вызвавшей ее средой реализуется через возвращаемое значение функции `main()`. Это значение может быть лишь целочисленным и чаще всего используется в качестве индикатора успешного завершения программы или номера ошибки.

Самым общим методом взаимодействия программы со средой является использование стандартных потоков ввода-вывода. Стандартная библиотека C++ предоставляет 3 таких потока ввода-вывода — `stdout`, `stdin` и `stderr`, которым соответствуют объекты `cout`, `cin` и `cerr`. Следует отметить, что вывод в поток `stdout` буферизуется, т.е. выводимые данные накапливаются в некоторой области памяти (в буфере) и отправляются в консоль лишь после заполнения буфера или при принудительном его опустошении ("flush"). Специальный мета-символ `std::endl` предназначен именно для опустошения буфера выходного потока. В противоположность этому стандартный поток ошибок не буферизируется.

Стандартные потоки ввода и вывода по умолчанию связываются с консолью. Но это необязательно. Все популярные командные интерпретаторы в среде Linux предоставляют средства для перенаправления потоков ввода-вывода. Рассмотрим следующую команду:

```
$ : program > outputfile.txt 2>&1
```

Запускается программа `program`, стандартный поток вывода которой перенаправляется в файл `outputfile.txt`, а стандартный поток ошибок (файловый дескриптор номер 2) объединяется со стандартным потоком вывода (файловый дескриптор номер 1). В результате работы этой команды на экран ничего не будет выведено.

Более интересным, однако, является следующее применение переадресации потоков ввода-вывода.

```
$ : program1 | program2
```

Этой командой запускается программа `program1`, вывод которого передается в качестве входного потока для программы `program2`. Такие конструкции очень удобны для составления более сложных программ из имеющихся простых.

Еще одной важной разновидностью взаимодействия является Environment (Среда). Это - коллекция предопределенных пар имен и значений, которые, однако, могут быть модифицированы программой. Каждая программа получает «по наследству» среду своего родительского процесса. Например, среду командного интерпретатора можно вывести на экран вызовом команды `set`. Там прописаны такие значения, как путь к домашнему каталогу пользователя, номер дисплея, переменная `$PATH` и т.д.

1.7. Обработка ошибок

При разработке приложений очень важно уделять внимание обработке исключительных ситуаций. Например, допустим, вы написали функцию, которая принимает только целые значения меньше ста. Поскольку такого типа в языке C++ нет, то нужно использовать ближайший по интервалу значений тип из доступных системных, например `unsigned int`. Тут возникает проблема, так как ваша функция может быть вызвана с любым значением аргумента из области значений типа `unsigned int`. Для предотвращения передачи некорректных значений компилятор не может провести никаких проверок, следовательно, проверку нужно сделать на уровне логики приложения.

Есть два класса ошибочных ситуаций, которые стоит различать в приложениях.

Во-первых, это ситуации, когда приложение получило на вход некорректные данные либо определило, что в среде (environment), где оно выполняется, что-то не так: нет прав доступа к нужным файлам, недостаточно памяти, сетевое соединение оказалось внезапно разорвано и т.п. В таких случаях стоит сообщить о выявленной проблеме (вывести сообщение в консоль, в системный журнал событий или ещё куда-то). Если продолжение работы из-за этой проблемы стало невозможным или нежелательным, приложению стоит завершить работу. При обработке таких ошибок нередко используются C++ exceptions и другие средства. Отметим, что такие ошибки не являются сами по себе ошибками разработчика данного приложения.

Во-вторых, бывают ситуации, когда компоненты приложения (объекты, функции и т.д.) используются в нём в условиях, когда это недопустимо. Например, в некоторой функции `foo(void *p)` предполагается, что ей передаётся указатель `p`, не равный `NULL`. Если в каких-то ситуациях происходит всё же вызов `foo(NULL)`, - это ошиб-

ка разработчика приложения. Для быстрого выявления таких ошибок нередко используется макрос `assert()`. Он принимает один аргумент - выражение - и, если это выражение равно 0, аварийно завершает работу приложения. `assert()` можно использовать и в окончательной версии приложений: если при компиляции приложения определён препроцессорный символ `NDEBUG`, `assert()` ничего не делает, если не определён - работает, как описано выше. Читателю рекомендуется активно пользоваться `assert()` для повышения безопасности приложений.

Другой проблемой являются системные вызовы. Всегда нужно иметь в виду, что каждый вызов системной функции может завершиться неудачей — начиная выделением памяти и созданием процесса и завершая системными вызовами, которые работают с внешними устройствами. Следовательно, после каждого системного вызова нужно проверять возвращаемое значение. Чаще всего системные функции при удачном завершении возвращают нулевое значение. При написании приложений нужно ознакомиться с руководствами соответствующих системных вызовов.

Возвращаемые ненулевые значения часто обозначают какую-то проблему. Для иных же функций ненулевое значение — это просто статус, а сам код ошибки находится в другом месте. В среде Linux для передачи кодов ошибок используется глобальная переменная `errno`, которая объявлена в заголовочном файле `<errno.h>`. Там же (или в одном из файлов, которые включены из файла `errno.h`) в частности можно найти определения и комментарии для всех возможных системных ошибок. Правильно написанный исходный код иногда по большей части состоит из проверок и обработки ошибок, чем из реализации самой логики программы.

1.8. Создание и использование библиотек

Как говорилось ранее, библиотеки в среде Linux бывают двух типов: динамические и статические. Статические библиотеки — это архивы, которые содержат несколько объектных файлов. Это аналог `.lib` файлов в Windows. При компоновке со статическим архивом компоновщик распаковывает архив и в его содержимом ищет подходящий файл. При нахождении копирует этот файл и статически ком-

понует с вашим приложением. Архивы можно создать при помощи утилиты ar следующим образом:

```
ar cr libtest.a obj1.o obj2.o
```

Эта команда запускает утилиту ar с параметром cr (create), задает имя выходного архива (libtest.a) и указывает, что нужно запаковать объектные файлы obj1.o и obj2.o.

Динамические библиотеки создаются самим GCC. Следующие несколько команд создают динамическую библиотеку с именем libtest.so, которая состоит из файлов test1.o и test2.o.

```
g++ -c -fPIC test1.cpp
g++ -c -fPIC test2.cpp
g++ -shared -fPIC -o libtest.so test1.o test2.o
```

Параметр -fPIC указывает компилятору на то, что код компилируется для динамической библиотеки (PIC — Position Independent Code).

В независимости от типа библиотеки для указания использования с приложением определенных библиотек компилятор нужно запускать с одними и теми же параметрами. Во-первых, необходимо передать каталог, в котором находятся библиотеки, и, во-вторых, перечислить нужные библиотеки.

```
g++ -o someapp someapp.o -L/usr/local/lib -ltest
```

Эта команда сообщает компилятору, что библиотеки нужно искать в каталоге /usr/local/lib (параметр -L/usr/local/lib) и нужно выбрать библиотеку с именем libtest (параметр -ltest). В том случае, когда в каталоге есть два варианта библиотеки libtest — libtest.so и libtest.a, то компилятор по умолчанию выбирает динамическую версию. Изменить это поведение можно, передав компилятору параметр -static.

1.9. Переносимость программ: Linux Standard Base

То обстоятельство, что операционная система Linux является открытой, влечет за собой определенные последствия, одним из которых является наличие огромного количества различных дистрибутивов и их версий, имеющих как сходства, так и различия. В мире насчитывается более 500 наименований дистрибутивов Linux (без учета

версий). Некоторые из этих систем являются базовыми, некоторые производными от других. В настоящее время основными базовыми дистрибутивами являются RedHat, SuSE, Mandriva, Debian, Ubuntu. Каждая из этих систем содержит свои особенности. Общими для всех систем являются ядро (kernel) и базовые системные утилиты — такие, как `cp`, `ls`, `mkdir`. Все они имеют во многом схожую структуру файловой системы, имеют более или менее схожие множества библиотек. Но, с другой стороны, они имеют разные подсистемы инсталляции программного обеспечения, разные инструменты настройки системы, но самое существенное для разработчиков приложений — что разные дистрибутивы имеют разный состав и версии библиотек, доступных по умолчанию для использования приложениями.

Даже когда имена библиотек одни и те же, это еще не дает гарантии, что они имеют одни и те же интерфейсы или что эти интерфейсы реализованы аналогичным образом. Это приводит к проблеме совместимости операционных систем семейства Linux. Дело в том, что многие библиотеки, которые используются разработчиками программного обеспечения, выпускаются под открытыми лицензиями, в частности, GPL (General Public License, www.gnu.org/licenses/gpl.html). Такие лицензии дают право на свободное изменение исходного кода. В результате, кроме естественной эволюции основных версий библиотеки, возникают различные модификации одной и той же базовой версии (например, для оптимизации производительности на аппаратном обеспечении определенного вида). Даже несмотря на то, что во время запуска приложение может проверять систему на наличие конкретной версии требуемой библиотеки, этого может быть недостаточно для его успешной работы.

Для минимизации подобных проблем и повышения совместимости операционных систем Linux и был создан проект LSB: Linux Standard Base (www.linuxbase.org). Цель LSB — определение стандартной среды для разработки приложений для системы Linux. Реализуется это следующим образом. Анализируются распространенные и известные приложения с целью выявления наиболее популярных библиотек. Далее эти библиотеки анализируются для выявления более часто используемых интерфейсов. Все эти интерфейсы наряду со всеми необходимыми типами, макросами и константами, которые, так или иначе, относятся к этим интерфейсам, собираются в единую базу. На основе такой базы решается, какие библиотеки и конкретные интерфейсы включать в «базовую среду» Linux (стандартизировать в

очередной версии стандарта). Стандартизованные библиотеки поддерживаются всеми основными дистрибутивами Linux (существует процесс сертификации соответствия дистрибутива стандарту).

Для входящих в стандарт библиотек генерируются специальные заголовочные файлы для использования при сборке переносимых приложений. Если разработчик приложений будет использовать данные заголовочные файлы, то в результате он создаст приложение, которое будет совместимым с LSB (будут автоматически использованы нужные версии интерфейсов). В свою очередь, это приведет к тому, что приложение будет одинаково компилироваться и выполняться на всех LSB совместимых дистрибутивах Linux. Конечно, для того чтобы можно было собрать приложение с такими заголовочными файлами, оно должно опираться только на тот состав библиотек и их функций, которые входят в стандарт LSB.

2. ИСПОЛЬЗОВАНИЕ СИСТЕМНЫХ ВЫЗОВОВ LINUX

В этой главе обсуждаются часто используемые системные объекты, системные вызовы и методы взаимодействия объектов. На их основе можно научиться создавать приложения любой сложности. В качестве примеров рассматриваются программы с интерфейсом командной строки (консольные приложения). В конце главы приводится пример, использующий изученные объекты и системные вызовы в реальном приложении.

2.1. Процессы

Одним из ключевых понятий в любой операционной системе являются *процессы*. Однако нужно отметить, что создание процесса в среде Linux в корне отличается от среды Windows.

2.1.1. Что такое процесс

Кратко и неформально можно сказать, что процесс — это программа во время выполнения. С другой точки зрения, процесс — это системный объект, который можно создать, которым можно манипулировать и который можно завершить или разрушить при помощи соответствующих системных вызовов. Процесс, как и любой другой системный объект, имеет уникальный идентификатор — `ID`. Получить идентификатор процесса можно, вызвав системную функцию `getpid()`. Идентификатор родительского процесса получают с помощью системного вызова `getppid()`.

Подробную информацию о запущенных в данный момент процессах в системе выдает утилита `ps`. Например, следующая команда отображает процессы, запущенные всеми пользователями в системе:

```
~ : ps ax
```

Завершить процесс можно вызовом утилиты `kill`, передавая идентификатор процесса, который нужно закрыть.

2.1.2. Создание процессов

Создание процесса в среде Linux имеет свою специфику. Единственная возможность создания процесса - системный вызов `fork()`. В результате вызова этой функции создается новый процесс, который представляет собой клон процесса, вызвавшего `fork()`. То есть после вызова `fork()` в системе получается два одинаковых процесса. Дальше каждый из процессов продолжает свое выполнение в соответствии со своей логикой. В частности, обычно вновь созданный процесс загружает какую-то другую программу, а родительский процесс продолжает свое выполнение или сразу ждет завершения запущенного процесса. Ожидание завершения запущенного процесса является обязательным, так как в противоположенном случае новый процесс после завершения превращается в «процесс-зомби».

«Зомби» — это процесс, который завершил свое выполнение, но не выгружен из оперативной памяти. Такие процессы загружают систему мусором и замедляют ее выполнение. Поэтому каждый процесс в ответе за создаваемые им новые процессы в плане освобождения ресурсов и нормального завершения.

Тут может возникнуть вопрос: если вновь созданный процесс является клоном создавшего его процесса, то как он может выполнить другие команды? Дело в том, что функция `fork()` имеет очень интересное поведение: она возвращает родительскому процессу идентификатор вновь созданного процесса, а последнему - нулевое значение. В листинге 2.1 показано, как можно использовать `fork()` для создания процесса.

Листинг 2.1: Применение fork()

```
1  #include <iostream>
2  #include <unistd.h>
3  #include <sys/wait.h>
4  #include <sys/types.h>
5  using namespace std;
6
7  int main()
8  {
9        cout<< "Parent process: forking." << endl;
10       pid_t ChildPid = -1;
11       ChildPid = fork();
12       if (ChildPid == -1)
13       {
```

```
14              cerr << "Cannot fork!!" << endl;
15              return -1;
16        }
17        if (ChildPid != 0) // Parent process
18        {
19              cout << "Parent process: waiting for
                child to terminate." << endl;
20              waitpid( ChildPid, 0, 0 );
21              cout << "Parent process: child
                process terminated. Exiting."<<endl;
22        }
23        else // Child process
24        {
25              cout << "Child process: doing some
                own job." << endl;
26        }
27        return 0;
28 }
```

Как видно из листинга, ожидание в родительском процессе производится вызовом функции `waitpid()`. Подробно о функциях ожидания можно прочесть в их руководстве пользователя: `man waitpid`.

Для загрузки программы в процесс используются так называемые `exec`-функции. Это семейство функций, имена которых начинаются на `exec`. Все они загружают программу, но есть различия. Их можно разделить на несколько групп.

- Функции, у которых в имени имеются буквы `p` или `l` (`execvp`, `execlp`), в качестве аргумента получают имя программы. Эту программу они ищут в путях, задаваемых переменной `$PATH`. Если же в имени функции нет этих букв, то в качестве аргумента им нужно передать полный путь программы.
- Функции, у которых в имени содержится буква `v` (`execv`, `execve`, `execvp`), принимают список параметров для программы как массив строковых значений, который завершается нулевым элементом. Если в имени функции содержится буква `l` (`execl`, `execlp`, `execle`), то список аргументов они получают при помощи механизма `varargs` языка C.
- Функции, у которых в имени содержится буква `e` (`execve`, `execle`), принимают еще один аргумент, в котором передается массив переменных среды.

Поскольку функции `exec` заменяют выполняемую программу новой, то они никогда не завершаются, кроме как в случае ошибок.

Листинг 2.2 показывает, как можно загрузить программу `ps`, передавая параметр `ax`.

Листинг 2.2: Применение fork() в паре с exec

```
1  #include  <iostream>
2  #include  <sys/wait.h>
3  #include  <sys/types.h>
4  #include  <unistd.h>
5  using namespace std;
6
7  int main ()
8  {
9     char* args[] = {
10     "ps", /* argv[0], the name of the program */
11     "ax",
12     NULL  /* The list must end with a NULL. */
13    };
14    pid_t child_pid = fork();
15    if (child_pid != 0)
16       return waitpid(child_pid, 0, 0);
17    else
18    {
19       execvp ("ps", args);
20       cerr   << "An error occurred in execvp"
                << endl;
21       return -1;
22    }
23    cout << "done with main program";
24    return 0;
25 }
```

2.1.3. Сигналы

В данном пособии будут описаны некоторые механизмы межпроцессного взаимодействия (Inter-Process Communication, IPC). Прежде всего следует ознакомиться с одним из основных механизмов — с *сигналами*. Сигнал — это простейший механизм передачи информации между процессами. Когда процесс получает какой-либо сигнал, то немедленно переходит к его обработке (нельзя считать, что строки

исходного кода или даже отдельные выражения являются атомарными операциями и не могут быть прерваны «посредине выполнения»). У каждого сигнала есть уникальный номер, вместо которого целесообразно использовать соответствующие определенные имена из файла `<signal.h>` (на самом деле они определены в файле `/usr/include/bits/signal.h`, но его напрямую включать не рекомендуется).

При получении сигнала процесс может действовать по-разному в зависимости от сигнала. Для некоторых сигналов поведение процесса предопределено системой, но это поведение можно изменить, скажем, установив обработчик или игнорируя сигнал.

Переопределение поведения при получении сигнала делается с помощью функции `sigaction`. Эта функция принимает три параметра. Первый параметр — номер сигнала, который нужно обрабатывать. Второй и третий параметры — указатели на структуру типа `sigaction`. Второй параметр содержит информацию о новом поведении процесса, в то время как третий параметр возвращает предыдущие значения. Самым важным полем структуры `sigaction` является `sa_handler`. Данное поле может иметь три вида значений:

- `SIG_DFL`, которое означает поведение по умолчанию;
- `SIG_IGN`, которое определяет, что сигнал должен быть проигнорирован;
- указатель на функцию-обработчик, которая должна иметь один параметр — значение сигнала — и возвращать `void`.

Обычно обработчики сигналов должны просто зарегистрировать факт получения сигнала и возвратить управление основной программе, так как из-за получения сигнала процесс может быть приостановлен в любом состоянии, что делает необходимым ограничить функциональность обработчика сигналов.

Однако есть сигналы, которые процесс не может игнорировать. Таким является сигнал `SIGKILL` (в отличие от сигнала `SIGTERM`, который может быть проигнорирован). Если утилита `kill`, о которой говорилось выше, запущена без параметров, она посылает сигнал `SIGTERM`.

В листинге 2.3 показан сценарий использования обработчика сигнала.

Листинг 2.3: Обработка сигнала SIGSEGV

```
1  #include <iostream>
2  #include <string.h>
3  #include <sys/types.h>
4  #include <signal.h>
5  using namespace std;
6
7  void signal_handler(int sig_num)
8  {
9        // Skip all signals except SIGSEGV
10       if (sig_num != SIGSEGV)
11             return;
12       cout << "Segmentation fault. Printing
                 stack trace" << endl;
13       // Some code to print out the stack trace
14 }
15
16 int main()
17 {
18       struct sigaction sa;
19       memset(&sa, 0, sizeof(struct sigaction));
20       sa.sa_handler = signal_handler;
21       sigaction(SIGSEGV, &sa, NULL);
22       int * InvalidPointer = 0;
23       cout << "Performing illegal operation"
              << endl;
24       *InvalidPointer = 100;
25       cout << "After handling SIGSEGV" << endl;
26       return 0;
27  }
```

Хотя приведенный в листинге код компилируется без ошибки, все равно при его запуске вы обнаружите странное поведение. Точнее говоря, процесс зайдет в бесконечный цикл. Это связано с тем, что на 24-й строке кода (`int * InvalidPointer = 0;`) происходит присваивание по адресу, которое находится вне адресного пространства процесса, что приводит к получению сигнала `SIGSEGV` (Segmentation Fault). Поскольку процесс регистрировал обработчик для этого сигнала, то, не дождавшись окончания выполнения операции присваивания, вызывается обработчик сигнала, который просто выводит сообщение. После завершения обработчика процессор возвращается к основному процессу, и поскольку регистр IP (Instruction Pointer) не был

обновлен, то снова выполняет ту же команду присваивания, что вновь приводит к сигналу, и так далее. С помощью этого примера можно более глубоко понять механизм сигналов в среде Linux. Таким образом, во избежание подобных ситуаций рекомендуется в обработчиках таких сигналов завершать процесс.

Поскольку сигнал — примитивный метод межпроцессного взаимодействия, то следует также упомянуть о механизме передачи системных сигналов между двумя пользовательскими процессами. Для этой цели используется функция `kill()`:

```
int kill (pid_t pid , int sig);
```

Эта функция посылает процессу с номером `pid` сигнал с номером `sig`.

2.2. Потоки

Поток (thread) — наименьший объект, которым может оперировать планировщик. То есть выполнение процесса на самом деле происходит через потоки. Каждый процесс содержит по крайней мере один поток (этот поток называют основным потоком процесса). Выполнение процесса начинается и заканчивается его основным потоком. Несколько потоков, принадлежащих одному и тому же процессу, разделяют секторы кода и данные его адресного пространства, но стек у каждого из потоков собственный. Это позволяет разным потокам оперировать общими данными, но находится в разных стадиях выполнения.

В системе Linux работа с потоками реализована в соответствии со стандартом POSIX (см. раздел "Posix Threads", также известный как "pthreads"). В стандарте описаны как типы и функции, предназначенные для создания и манипуляции потоками, так и различные объекты и механизмы синхронизации потоков.

2.2.1. Создание потока

Согласно стандарту POSIX, поток создается функцией `pthread_create`, которая определена в файле `<pthread.h>` следующим образом:

```
int pthread_create (pthread_t *,
                    pthread_attr_t *,
                    void * (*) (void *),
                    void *)
```

Первый параметр — указатель на `pthread_t`, куда `pthread_create` возвращает дескриптор вновь созданного потока. Второй параметр задает атрибуты потока, но его здесь мы рассматривать не будем. Третий параметр — указатель на функцию, о котором говорится чуть ниже. Последний параметр — данные, которые могут быть переданы функции из третьего параметра (см. 2.2.2).

При создании потока для начала необходимо определить функцию, с выполнения которой и начнется поток (так называемую «поточную функцию» или «функцию потока»). Эта функция должна иметь определенную сигнатуру:

```
void * thread_func (void *)
```

Поток завершает свою работу, когда он доходит до последней команды заданной ему функции. В листинге 2.4 иллюстрировано создание POSIX потока в среде Linux.

Листинг 2.4: Создание потока

```
1   #include <iostream>
2   #include <pthread.h>
3   using namespace std;
4
5   void * thread_func(void *)
6   {
7         while (true)
8               cerr << '-';
9         return 0;
10 }
11
12 int main()
13 {
14        pthread_t thread;
15
16        pthread_create(&thread, 0,
                         &thread_func, 0);
17
18        while (true)
19              cerr << '|';
```

```
20
21        return 0;
22 }
```

При компиляции такого исходного файла нужно указать на библиотеку `pthread` следующим образом:

```
g++ thread_create.cpp –o thread_create –lpthread
```

После запуска программы на экране поочередно и хаотично будут появляться символы «—» и «|» (вертикальные и горизонтальные линии). Это свидетельствует о том, что оба потока работают параллельно.

Как уже говорилось, процесс завершает свою работу, когда завершается его основной поток. Следовательно, основной поток процесса должен ожидать завершения рабочих потоков. Данную функциональность предоставляет функция

```
int pthread_join (pthread_t, void **)
```

Первый параметр — дескриптор потока, который нужно ждать, а второй параметр — указатель, в котором возвращается значение, возвращаемое поточной функцией.

2.2.2. Передача данных

Иногда вновь созданный поток нуждается в каких-либо данных для выполнения. Например, необходимо узнать свой порядковый номер. Аргумент функции потока предназначен именно для этой цели. Однако, как вы, возможно, заметили, поточная функция может принимать лишь один параметр типа `void*`. Полезно ознакомиться с этим типом данных поближе. Тип `void*` можно охарактеризовать как универсальный транспорт для передачи данных. Дело в том, что указатель на любой тип можно привести к типу `void*` и наоборот. То есть для любого типа `T` разрешается написать следующее:

Примечание 2.2.1. Указатель типа `void *` не подлежит разыменованию (dereferencing).

```
T object;
void * v;
v = ( void * )&object;
object = * (T* ) v;
```

Это свойство позволяет передавать поточным функциям любые объекты. Для наглядности рассмотрим следующую задачу. Задан массив целых чисел `array` длины `N`. Требуется написать многопоточную программу, которая вычисляет сумму его элементов. При этом каждый поток вычисляет сумму какой-то части массива. Результат нужно записать в глобальную переменную `GlobalSum`. В листинге 2.5 приведено решение этой задачи.

Примечание 2.2.2. Следует отметить, что в приведенном ниже примере не обсуждается вопрос синхронизации. Приведенное решение чревато скрытыми ошибками, которые могут проявляться лишь спустя некоторое время исполнения. Подробнее о синхронизации см. 2.2.4.

Листинг 2.5: Передача данных

```
1  #include <iostream>
2  #include <pthread.h>
3  using namespace std;
4
5  const int N = 10000000;
6  const int ThreadCount = 25;
7  int GlobalSum = 0;
8  int array[N];
9
10 void * thread_func(void * param)
11 {
12       int ThreadNumber = *(int*)param;
13       int LocalSum = 0;
14       for (int i = N/ThreadCount * ThreadNumber;
              (i < N/ThreadCount *
                 (ThreadNumber + 1)) && (i < N);
              ++i)
15       {
16             LocalSum += array[i];
17       }
18       GlobalSum += LocalSum;
19       return NULL;
20 }
21
22 int main()
23 {
24       for (int i = 0 ; i < N; ++i)
```

```
25              array[i] = 1;
26        pthread_t Threads[ThreadCount];
27        for (int i = 0; i < ThreadCount; ++i)
28        {
29              int * ThreadNumber = new int;
30              *ThreadNumber = i;
31              pthread_create(&Threads[i], NULL,
                  thread_func, (void*)ThreadNumber);
32        }
33        for (int i = 0; i < ThreadCount; ++i)
34        {
35              pthread_join(Threads[i], NULL);
36        }
37        cout << "GlobalSum : " << GlobalSum
               << endl;
38        return 0;
39  }
```

Внимательный читатель мог заметить странность на строках 29–30. Значение счетчика цикла копируется в указатель, который позднее передается поточной функции. Дело в том, что если напрямую передать адрес счетчика, то из-за задержки при старте потока поточная функция на самом деле получит уже измененное значение.

Еще одной странностью может показаться то, что в поточной функции подсумма просчитывается в локальной переменной и только после того, как сумма готова, она добавляется к глобальной сумме. Это сделано с расчетом на будущую синхронизацию данного кода.

2.2.3. Завершение потока

Как правило, поток завершается, когда доходит до конца выполнения основной функции этого потока. Однако иногда возникает потребность в досрочном завершении потока другим потоком. Такое явление обозначается термином *thread cancellation*. Принудительное завершение потока может привести к появлению «мусора» (например, незакрытых дескрипторов или выделенной памяти). Поэтому поток может определить, что завершаться извне не может. По этому признаку потоки делятся на несколько видов.

- *Асинхронно завершаемые потоки*. Такой поток может быть завершен в любой момент.
- *Синхронно завершаемые потоки*. Такой поток может быть завершен лишь в каких-то определенных точках.

- *Незавершаемые потоки*. Такой поток нельзя завершать извне. Все запросы на завершение потока просто игнорируются.

Регулировать такое поведение потоков можно при помощи функций: `pthread_setcanceltype` и `pthread_setcancelstate` (подробно об этих функциях можно прочесть в руководстве пользователя: `man pthread_setcancelstate`). Функция `pthread_setcancelstate` позволяет потокам реализовать критические секции, то есть код, который должен либо выполняться полностью, либо вообще не выполняться. Об этом речь будет идти в 2.2.4.

2.2.4. Синхронизация и критические секции

Как уже говорилось, в программе, приведенной в листинге 2.5, есть скрытая проблема. Рассмотрим исходный код под микроскопом, то есть будем анализировать код Ассемблера. Для получения кода на языке Ассемблер, нужно запустить следующую команду:

```
g++ –S thread_data nonsync.cpp
```

В результате создается файл `thread_data_nonsync.s`, который и содержит требуемый код. В листинге 2.6 приведен отрезок кода, соответствующий строке 18 из листинга 2.5.

Листинг 2.6: Код на Ассемблере

```
1  movl   GlobalSum(%rip), %eax
2  addl   -8(%rbp), %eax
3  movl   %eax, GlobalSum(%rip)
```

Как видно, простейший оператор `+=` на самом деле состоит из нескольких команд Ассемблера. Следовательно, при выполнении этого оператора может произойти прерывание, и планировщик передаст управление другому потоку или процессу. Следовательно, если два потока, которые считают сумму частей массива, одновременно дойдут до строки 18, то один из них может «помешать» другому. А именно: в тот момент, когда первый поток уже загрузил значение `GlobalSum` в регистр `EAX` и добавил к нему значение своей `LocalSum`, может произойти прерывание, и второй поток перезапишет значение регистра `EAX` значением `GlobalSum`, этим самым потеряв вычислен-

ную сумму первого потока. Следовательно, в результате получится неправильная сумма.

Когда, в зависимости от того, в какой очереди выполняются потоки или процессы, может изменяться результат, говорят, что имеют место *условия состязания* (race conditions). В основном такие условия возникают, когда существует ресурс, распределенный между несколькими процессами или потоками. Решением этой проблемы является синхронизация. Синхронизация обычно реализуется путем предотвращения одновременного доступа к ресурсу более чем одним потоком или процессом.

Мьютексы

Одним из объектов синхронизации являются *мьютексы* (mutex: Mutual Exclusion — взаимное исключение). Это — объект, который имеет два состояния: занят или свободен. Если кто-либо уже занял его, то все повторные попытки переводят поток к приостановленному состоянию. Как только мьютекс освобождается, поток, который ждал его, перезапускается, и мьютекс передается ему.

Мьютекс в среде Linux представляется типом `pthread_mutex_t`, который можно создать функцией `pthread_mutex_init` или вызовом макроса `PTHREAD_MUTEX_INITIALIZER`. Для захвата и освобождения мьютекса используются соответственно функции `pthread_mutex_lock` и `pthread_mutex_unlock`. В листинге 2.7 показано решение предыдущей задачи с использованием мьютексов для синхронизации потоков.

Листинг 2.7: Использование мьютексов

```
1  #include <iostream>
2  #include <pthread.h>
3  using namespace std;
4
5  const int N = 10000000;
6  const int ThreadCount = 25;
7  int GlobalSum = 0;
8  int array[N];
9  pthread_mutex_t GlobalMutex;
10
11 void * thread_func(void * param)
12 {
```

```
13        int ThreadNumber = *(int*)param;
14        int LocalSum = 0;
15        for (int i = N/ThreadCount *
               ThreadNumber;
              (i < N/ThreadCount*(ThreadNumber+1))
               && (i < N); ++i)
16        {
17              LocalSum += array[i];
18        }
19        pthread_mutex_lock(&GlobalMutex);
20        GlobalSum += LocalSum;
21        pthread_mutex_unlock(&GlobalMutex);
22        return NULL;
23 }
24
25 int main()
26 {
27        pthread_mutex_init(&GlobalMutex, NULL);
28        for (int i = 0 ; i < N; ++i)
29              array[i] = 1;
30        pthread_t Threads[ThreadCount];
31        for (int i = 0; i < ThreadCount; ++i)
32        {
33              int * ThreadNumber = new int;
34              *ThreadNumber = i;
35              pthread_create(&Threads[i], NULL,
                  thread_func, (void*)ThreadNumber);
36        }
37        for (int i = 0; i < ThreadCount; ++i)
38        {
39              pthread_join(Threads[i], NULL);
40        }
41        cout <<"GlobalSum : " <<GlobalSum <<endl;
42        pthread_mutex_destroy(&GlobalMutex);
43        return 0;
44 }
```

Поскольку мьютекс — это средство блокирования потока другим потоком, то могут возникать так называемые тупиковые блокировки (deadlocks). Тупиковая блокировка — это ситуация, когда поток (или потоки) ждет чего-то, что никогда не может произойти. Поток может блокировать сам себя, скажем, вызвав `pthread_mutex_lock` дважды на одном и том же мьютексе типа «быстрый мьютекс» (fast mutex —

это тип мьютексов по умолчанию). Если же мьютекс имеет тип «рекурсивный», то повторный захват не приведет к тупику. Тип мьютекса можно задать при инициализации, передавая функции `pthread_mutex_init` объект типа `pthread_mutex_attr_t`.

Семафоры

Семафор — это тоже объект синхронизации, который, в отличие от мьютекса, может иметь несколько значений. Для семафора определены операции увеличения значения (`post`) и операция уменьшения значения (`wait`), причем значение семафора не может быть отрицательным. Следовательно, при вызове `wait` на нулевом значении семафора поток переходит в приостановленное состояние, пока какой-то другой поток не увеличит значение семафора.

При использовании семафоров нужно включить в программу заголовочный файл `<semaphore.h>`. Семафоры в среде Linux представлены типом данных `sem_t`. Для инициализации семафора используется функция `sem_init`, где первый параметр — указатель на семафор, который нужно инициализировать. Второй параметр всегда должен быть нулевым, а третий задает начальное значение семафора. Операции увеличения и уменьшения значения семафора — соответственно `sem_post` и `sem_wait`. Более подробно о семафорах речь будет идти в 2.3.4.

Условные переменные

Условная переменная — это объект, который может находиться в двух состояниях: либо условие имеет место, либо нет. Это используется, когда поток должен подождать, пока определенное условие не будет иметь место. В таком случае поток, который инициирует это условие, должен подать знак ожидающим потокам. Условные переменные в среде Linux представлены типом данных `pthread_cond_t`. Инициализировать условную переменную можно вызвав функцию `pthread_cond_init`. Для ожидания выполнения условия используется функция `pthread_cond_wait`, а для оповещения — функция `pthread_cond_signal`. Как показывает опыт, при ожидании какого-либо условия обычно появляется проблема синхронизации. Для этого условные переменные используются в паре с мьютексами. Это реализовано следующим образом: функция `pthread_cond_wait` в качестве второго параметра принимает объект типа `pthread_mutex_t*`. Этот

мьютекс освобождается перед тем, как поток начинает ожидание. Как только ожидание завершается, поток захватывает мьютекс, и только после этого функция `pthread_cond_wait` завершает свое выполнение. То есть в то время, когда поток ожидает выполнения условия, мьютекс свободен. В листинге 2.8 показан сценарий использования условных переменных в паре с мьютексами при решении следующей задачи. Нужно запустить два потока, один из которых откуда-то получает какие-то сообщения, помещает их в определенный разделяемый буфер и подает сигнал второму потоку, который берет сообщение из буфера и использует по назначению.

Листинг 2.8: Использование условных переменных

```
1  #include <stdlib.h>
2  #include <iostream>
3  #include <pthread.h>
4  using namespace std;
5
6  int Buffer;
7  pthread_mutex_t BufferMutex;
8  pthread_cond_t MessageArrived;
9
10 void * MessageReceiver(void * param)
11 {
12        int LocalMessage = 0;
13        while (true)
14        {
15              // Receive the message
16              sleep(rand() % 5);
17              cout << getpid() << ": Message
                        received." << endl;
18              pthread_mutex_lock(&BufferMutex);
19              Buffer = LocalMessage;
20              cout << getpid() << ": Signalling
                        message processor" << endl;
21              pthread_cond_signal
                      (&MessageArrived);
22              pthread_mutex_unlock(&BufferMutex);
23        }
24        return NULL;
25 }
26
27 void * MessageProcessor(void * param)
28 {
```

```
29        int LocalMessage = 0;
30        while (true)
31        {
32              pthread_mutex_lock(&BufferMutex);
33              cout << getpid() << ":\tWaiting for
                        message to arrive" << endl;
34              pthread_cond_wait(&MessageArrived,
                                  &BufferMutex);
35              cout << getpid() << ": \tCopying
                       message to start processing"
                     << endl;
36              LocalMessage = Buffer;
37              pthread_mutex_unlock(&BufferMutex);
38              // Process the message
39              sleep(rand() % 5);
40        }
41        return NULL;
42 }
43
44 int main()
45 {
46        pthread_mutex_init(&BufferMutex, NULL);
47        pthread_cond_init(&MessageArrived, NULL);
48
49        pthread_t ReceiverThread;
50        pthread_t HandlerThread;
51
52        pthread_create(&ReceiverThread,
                      NULL, MessageReceiver, NULL);
53        pthread_create(&HandlerThread,
                      NULL, MessageProcessor, NULL);
54
55        pthread_join(HandlerThread, NULL);
56        pthread_join(ReceiverThread, NULL);
57
58        pthread_mutex_destroy(&BufferMutex);
59        pthread_cond_destroy(&MessageArrived);
60        return 0;
61 }
```

2.3. Межпроцессное взаимодействие

На практике часто возникает надобность передачи данных или сообщений между несколькими процессами. И все операционные системы предоставляют некоторый механизм для решения такого рода задач. Этот механизм называют *межпроцессным взаимодействием* (IPC — Inter Process Communication). Реализуется он посредством нескольких объектов и соответствующих им системных вызовов.

В этой части пособия мы попробуем разъяснить основные механизмы межпроцессного взаимодействия и покажем, как они могут быть использованы в приложениях.

2.3.1. Разделяемая (общая) память

Одна из самых простых возможностей реализации межпроцессного взаимодействия — это разделяемая (общая) память. Разделяемая память — это объект операционной системы, который находится вне адресных пространств процессов, но может быть подключен к нескольким процессам, таким образом создавая участок памяти, который используется несколькими процессами.

> *Примечание 2.3.1.* Когда некоторый объект используется совместно несколькими процессами, то могут возникнуть условия состязания (race conditions), для обхождения которых необходимо использовать механизмы синхронизации.

Разделяемая память в системе Linux представляется типом данных `int`, который можно инициализировать вызовом функции `shmget()`:

```
int shmget(key_t key , size_t size , int shmflg)
```

Первый параметр данной функции — это некоторый идентификатор, известный процессам, которые будут далее взаимодействовать. В том случае, когда эти процессы «находятся в родстве» (один из них является родительским для другого, или же у них общий родительский процесс), то можно использовать системное значение `IPC_PRIVATE`. Это макрос, который при каждом вызове возвращает уникальное значение. Следовательно, «неродственные» процессы при использовании макроса `IPC_PRIVATE` получат разные объекты разделяемой памяти. Второй параметр — размер участка памяти, который на самом деле будет округлен вверх до ближайшего кратного размера

страницы (page size). Третий параметр задает флаги создания. Например, можно указать, что объект нужно создать (флаг IPC_CREAT), или в том случае, когда объект с этим ключом уже существует, то нужно возвращать ошибку (флаг IPC_EXCL). Помимо флагов такого рода, существуют и флаги, которые регулируют доступ к памяти для других процессов (например, флаг S_IRUSR указывает на то, что процессы этого пользователя имеют доступ на чтение этого участка памяти). Различные возможные значения флага можно найти в заголовочном файле <sys/stat.h>. Приведенный ниже код создает разделяемую память размером в один килобайт, который может быть использован «родственными» процессами на запись и на чтение.

```
int shm_id = shmget(IPC_PRIVATE, 1024,
                    IPC_CREAT | S_IRUSR | S_IWUSR);
```

После того как объект успешно создан, процесс, который хочет его использовать, должен прикрепить (attach) этот участок памяти к своему адресному пространству. Делается это при помощи функции shmat():

```
void * shmat(int shmid, const void *shmaddr,
             int shmflg);
```

Первый параметр — это дескриптор участка памяти, который должен быть прикреплен. Второй параметр — это адрес, куда, в случае возможности, участок должен быть прикреплен. Обычно этот параметр лучше оставить нулевым, тем самым позволяя операционной системе самой выбрать адрес. Третий параметр задает флаги, регулирующие режим прикрепления. Например, процесс может указать, что этот участок он будет использовать только для чтения (флаг SHM_RDONLY). В случае успеха функция возвращает адрес, который соответствует разделяемой памяти. В противном случае она возвращает значение -1. После удачного прикрепления возвращенный указатель нужно привести к требуемому типу и использовать для чтения или записи данных.

Когда блок разделяемой памяти больше не используется, процесс должен отсоединить (detach) его. Делается это при помощи функции shmdt():

```
int shmdt( const void *shmaddr);
```

В случае успеха функция возвращает нулевое значение. В случае ошибки она возвращает `-1`.

Когда участок разделяемой памяти больше не используется, создавший его процесс должен явно отсоединить такой участок и пометить его как свободный. После того как последний процесс отсоединит участок от своего адресного пространства, операционная система освободит память и разрушит объект. Делать это также можно при помощи функции `shmctl()`:

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

Примечание 2.3.2. Функция `shmctl` используется и для других целей. Например, с ее помощью можно получить статус объекта или поменять какие-то атрибуты. Для деталей можно обратиться к руководству пользователя: `man shmctl`.

Первый параметр — это дескриптор объекта разделяемой памяти. Второй параметр — это команда, которую нужно выполнить. Для удаления участка нужно задать значение `IPC_RMID`. Третий параметр используется для получения статуса или для определения атрибутов объекта. При удалении третий параметр должен иметь нулевое значение.

Листинг 2.9 иллюстрирует сценарий использования разделяемой памяти.

Листинг 2.9: Использование разделяемой памяти

```
1  #include <unistd.h>
2  #include <sys/stat.h>
3  #include <sys/types.h>
4  #include <iostream>
5  #include <sys/shm.h>
6  #include <string.h>
7
8  using namespace std;
9
10 int main()
11 {
12       int ShmID = shmget(IPC_PRIVATE, 1024,
                    IPC_CREAT | S_IRUSR | S_IWUSR);
13       if (ShmID == -1)
14       {
```

```
15              cerr << "Cannot create shared
                        memory" << endl;
16              return -1;
17        }
18        pid_t ChildPid = fork();
19        if (ChildPid == -1)
20        {
21              cerr << "Cannot fork" << endl;
22              return -2;
23        }
24        char * Buffer = (char*)shmat(ShmID, 0, 0);
25        if (Buffer == 0)
26        {
27              cerr << "Cannot attach shared
                        memory" << endl;
28              return -3;
29        }
30        if (ChildPid == 0)
31        {
32              // Child process. Write
                   something to the shared memory
33              strcpy(Buffer, "Hello");
34              shmdt(Buffer);
35              return 0;
36        }
37        else
38        {
39              //Parent proces. Wait a little bit
                and print the shared memory contents
40              sleep(1);
41              cout << "Shared memory contents: "
                     << Buffer << endl;
42              shmdt(Buffer);
43              shmctl(ShmID, IPC_RMID, 0);
44        }
45        return 0;
46  }
```

Примечание 2.3.3. Как говорилось выше, такого рода задачи должны быть синхронизированы, о чем будет идти речь в 2.3.4.

2.3.2. Каналы

Канал — это виртуальное устройство, имеющее два конца, один из которых — вход, а второй — выход. Хотя с точки зрения программиста виртуальные и реальные устройства не различаются, так как в среде Linux все устройства представляются файлами. В каталоге `/dev` можно найти файлы, которые соответствуют устройствам в системе. В определенном смысле канал — тоже файл. Этот файл может быть открыт сразу двумя процессами, один из которых открывает его только для чтения, а второй — только для записи. Каналы бывают двух типов. Один предназначен для использования только между родственными процессами (`pipe`), а второй может быть совместно использован любыми процессами, которым известно имя канала (fifo — First In First Out). Мы рассмотрим оба варианта каналов.

Pipe

Первый и более простой вариант каналов часто используется пользователями Linux, возможно, без понимания, что это и есть тот самый канал. Например, когда вы запускаете команду

```
ls | more
```

то указываете оболочке (`shell`) создать канал, вход которого должен быть прикреплен к стандартному потоку вывода процесса `ls`, а выход — к стандартному потоку ввода процесса `more`. В результате `ls` печатает содержимое текущего каталога в стандартный поток вывода, не зная, что на самом деле это стандартный поток ввода процесса `more`. Получается, что `more` во входе получает листинг каталога, который и по порциям выводит в стандартный поток вывода (который тоже в свою очередь может быть привязан к чему-либо еще).

Для создания канала нужно создать массив из двух целочисленных элементов, который передается функции `pipe()`. Она создает канал: в первый элемент переданного массива записывает файловый дескриптор выхода, а во второй элемент — дескриптор входа. Обычно после вызова `pipe()` используется `fork()`, и в результате оба процесса получают копии дескрипторов. После этого процессы в зависимости от их роли должны закрыть тот или иной дескриптор. Дальше процессы могут записать и прочитать данные через канал, как если бы это был простой файл. В листинге 2.10 показано, как именно это можно сделать.

Листинг 2.10: Использование каналов

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6         int fds[2];
7         if (pipe(fds) == -1)
8         {
9               cerr << "Cannot create pipe."
                     << endl;
10              return -1;
11        }
12
13        if (fork())
14        {
15              // Parent process.
                   Close the read end.
16              close(fds[0]);
17              write(fds[1], "asdasd", 7);
18              close(fds[1]);
19        }
20        else
21        {
22              close(fds[1]);
23              char buf[100];
24              read(fds[0], buf, 100);
25              cout << "Buffer contents: "
                     << buf << endl;
26              close(fds[0]);
27        }
28        return 0;
29 }
```

Альтернативно канал может быть создан функцией `popen()`.

```
FILE *popen(const char *command, const char *type);
```

Первый параметр — это команда, которую нужно запускать (имя программы и аргументы), а второй — режим открытия канала. Если нужно прочесть вывод какой-то программы, то во втором параметре нужно задать ”`r`”. В этом случае стандартный поток вывода команды, задаваемый первым параметром, присоединяется к входу канала, и в

результате основной процесс может «прочесть» вывод запущенного процесса. Если же во втором параметре задать значение ”w”, то выход созданного канала соединится со стандартным потоком ввода команды, задаваемым первым параметром. Все написанное в канал будет прочитано вновь созданным процессом из стандартного потока ввода.

Приведенный в листинге 2.11 пример показывает, как можно получить содержимое текущего каталога при помощи функции `popen()`.

Листинг 2.11: Использование каналов — popen

```
1  #include <iostream>
2  #include <stdio.h>
3  using namespace std;
4
5  int main()
6  {
7         FILE * Input = popen("ls -al", "r");
8
9         if ( Input == NULL )
10        {
11              cerr << "Cannot create pipe."
                     << endl;
12              return -1;
13        }
14
15        char buf[100];
16        fread(buf, 100, 1, Input);
17        cout << "Buffer contents: " << buf
                << endl;
18        pclose(Input);
19
20        return 0;
21 }
```

FIFO

Объект `FIFO` в Linux представляет собой файл в файловой системе. При создании задается путь и атрибуты доступа. Перед тем как приступить к программированию, имеет смысл для начала визуально посмотреть на `FIFO` в деле. Для этого в домашнем каталоге пользователя запустите команду `mkfifo test.fifo`. Потом нужно задать атрибуты доступа: `chmod +rw test.fifo`. После этого необходимо

запустить еще одну консоль. В первой консоли запустите команду `cat > test.fifo`, а во второй `cat < test.fifo`. Теперь если что-нибудь написать в первой консоли, то оно появится во второй.

Для создания объекта FIFO программным путем используется функция `mkfifo()`:

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

Первый параметр — это путь файла, а второй задает атрибуты доступа. Для нормальной работы в параметре `mode` нужно задать `S_IRUSR | S_IWUSR`. После успешного создания даже неродственные процессы могут открыть `FIFO`: один для записи, а другой для чтения и начать передачу данных.

2.3.3. Сокеты

Сокет — это устройство, которое имеет два конца, и оба могут быть использованы как для чтения, так и для записи. При использовании сокетов один из процессов обычно играет роль сервера, а остальные — клиентов. Сервер — это процесс, который предоставляет какую-то информацию или сервис. Клиенты — это пользователи сервера. С этой точки зрения сокеты тоже разделяются на серверные и клиентские. Для создания *серверного сокета* требуется определенная последовательность шагов.

- Создание сокета (функция `socket()`).
- Привязка сокета к адресу и порту (функция `bind()`).
- Перевод сокета в режим ожидания (функция `listen()`).
- Принятие входящего соединения (функция `accept()`).
- Передача и чтение данных (функции `send()`, `sendto()`, `recv()` и `recvfrom()`).
- Закрытие соединения (функции `shutdown()` и `close()`).

Для создания *клиентского сокета* необходимо меньшее количество действий.

- Создание сокета (функция `socket()`).
- Присоединение к серверу (функция `connect()`).
- Передача и чтение данных (функции `send()`, `sendto()`, `recv()` и `recvfrom()`).
- Закрытие соединения (функции `shutdown()` и `close()`).

Рассмотрим перечисленные функции более детально.

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

В случае успеха функция возвращает дескриптор созданного сокета. Первый параметр — домен сокета. Существует несколько доменов, например, AF_UNIX, AF_INET, AF_INET6, AF_IPX и т.д. В примерах мы будем использовать либо AF_UNIX, либо AF_INET. Домены AF_UNIX или AF_LOCAL — это специальные файлы (адресуются как обычные, но являются системными). Они используются лишь для передачи данных в рамках одной машины. Остальные перечисленные домены представляют сетевые сокеты. В частности, AF_INET — это сокет, предназначенный для коммуникации в сети IPv4, в то время как домен AF_INET6 предназначен для сетей с поддержкой IPv6.

Второй параметр — это тип сокета. Наиболее часто используемыми являются типы SOCK_STREAM и SOCK_DGRAM. Сокет типа SOCK_STREAM по своей функциональности схож с телефонным звонком. Сервер ожидает соединения. Клиент подсоединятся, получает подтверждение, потом для каждого отосланного пакета при получении адресатом приходит подтверждение. В случае неудачной отправки пакет пересылается повторно. То есть поточный сокет дает гарантии того, что каждый отосланный пакет будет получен адресатом либо отправляющая сторона получит оповещение об ошибке. Разумеется, такая надежность работы сокетов приводит к затратам во времени. Другим подходом являются дейтаграмм-сокеты. При отправке пакета через такой сокет отправитель не оповещается ни при удаче, ни при неудаче. Такой сокет работает очень быстро, но нет никакой гарантии надежности доставки. Если отправляющая сторона в течение какого-то времени не получает ответа, то ее придется запросить снова вручную. Читатель может найти схожесть между функциональностью дейтаграммных сокетов и почты.

Третий параметр задает протокол сокета. Для поточных сокетов (SOCK_STREAM) протоколом по умолчанию является TCP/IP (Transmission Control Protocol/Internet Protocol). Для дейтаграмм сокетов (SOCK_DGRAM) таковым является протокол UDP (User Datagram Protocol). Следовательно, следующая команда создает сетевой сокет, который работает по протоколу TCP/IP:

```
socket (AF_INET, SOCK_STREAM, 0);
```

Рассмотрим функцию `bind()`.

```
int bind (int socket,
          const struct sockaddr *address,
          socklen_t address_len);
```

Первый параметр — это дескриптор сокета для привязки. Второй параметр — указатель на структуру, в которой задаются атрибуты адреса (адресное семейство, адрес и порт). Третий параметр — это размер структуры, указатель на которую передан во втором параметре. Самым сложным в этом шаге является создание экземпляра типа `struct sockaddr`. На самом деле это «базовый» тип для нескольких структур, представляющих адрес для разных доменов. В случае сетевых сокетов `AF_INET` используется тип `sockaddr_in`, который определен следующим образом:

```
struct sockaddr_in
{
  /* socket family (domain) */
  sa_family_t  sin_family;

  /* Port number */
  in_port_t    sin_port;

  /* Internet address.*/
  struct in_addr sin_addr;
};
```

Поле `sin_family` нужно инициализировать значением `AF_INET`. Для задания порта требуется для начала привести число к сетевому представлению (порядок байтов может быть другим). Делается это вызовом функции `htons()` (host to network short). Для инициализации поля `sin_addr` нужно сначала получить IP-адрес из имени машины. Его можно получить при помощи функции `inet addr()`. В листинге 2.12 показан фрагмент исходного кода, где инициализируется объект типа `sockaddr_in`.

Листинг 2.12: Инициализация sockaddr_in

```
1  #include <sys/socket.h>
2  #include <netinet/in.h>
3  #include <arpa/inet.h>
4
5  struct sockaddr_in addr;
6  memset(&addr , 0 , sizeof (struct sockaddr_in);
7
8  addr.sin_family = AF_INET;
9  addr.sin_addr.s_addr = inet_addr(”127.0.0.1”);
10 addr.sin_port = htons(15678);
```

После того как сокет успешно привязан к адресу и номеру порта, нужно перевести его в режим ожидания соединения. Это делается вызовом функции `listen()`.

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

Первый параметр, — как обычно, дескриптор сокета, а второй — максимальное количество поддерживаемых соединений. Если это количество не нужно ограничивать, то можно передать нулевое значение.

После того как сокет успешно переведен в режим ожидания, вызывается функция `accept()`, которая в случае успеха возвращает новый сокет — клиентский сокет. Именно через этот сокет и производится обмен данными.

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

Первый параметр — это дескриптор сокета, а второй и третий параметры предназначены для получения информации об адресе клиента. Если эта информация не важна, то можно передать нулевые указатели.

Клиентский процесс тоже должен создать сокет, после чего он может подсоединиться к серверу. Делается это при помощи функции `connect()`.

```
int connect (int sockfd,
             const struct sockaddr *addr,
             socklen_t addrlen);
```

Первый параметр — это дескриптор сокета. А второй и третий параметры имеют то же значение, что и для функции `bind()`, лишь с той разницей, что тут нужно задать адрес сервера.

После того как клиент успешно подсоединился к серверу и сервер получил сокет клиента, они могут начать передачу данных. Делается это при помощи функций `send()` и `recv()`.

```
ssize_t send (int sockfd, const void *buf,
              size_t len, int flags);
ssize_t recv (int sockfd, void *buf,
              size_t len, int flags);
```

Когда сокет больше не используется, его следует «выключить» и закрыть. Делается это при помощи функций `shutdown()` и `close()`:

```
int shutdown (int socket , int how);
int close (int fildes);
```

Функции `shutdown()` нужно передать дескриптор сокета и указать, какую именно функциональность выключить: чтение, запись или обе (`SHUT_RD`, `SHUT_WR` и `SHUT_RDWR` соответственно). После того как сокет выключен, необходимо закрыть его файловый дескриптор вызовом `close()`.

Обе из этих функций получают дескриптор сокета, буфер, размер буфера и флаги. Возвращают они количество прочтенных или отправленных байтов соответственно.

Более подробно обо всех этих функциях можно прочесть в соответствующих руководствах пользователя. Наконец, в листингах 2.13 и 2.14 приведены исходные коды простых сервера и клиента соответственно. Детальный анализ этого кода позволит читателю лучше понять механизм использования сокетов.

Листинг 2.13: Пример сервера

```
1  #include <iostream>
2  #include <sys/socket.h>
3  #include <netinet/in.h>
4  #include <arpa/inet.h>
5  #include <string.h>
6
7  using namespace std;
8
9  int main()
```

```
10 {
11       int server_socket = socket(AF_INET,
                                    SOCK_STREAM, 0);
12       if (server_socket == -1)
13       {
14             cerr << "Cannot create socket"
                    << endl;
15             return -1;
16       }
17       sockaddr_in addr;
18       memset(&addr, 0, sizeof(sockaddr_in));
19       addr.sin_family = AF_INET;
20       addr.sin_addr.s_addr =
                           inet_addr("127.0.0.1");
21       addr.sin_port = htons(15678);
22       if (bind(server_socket, (sockaddr*)&addr,
            sizeof(sockaddr_in)) == -1)
23       {
24             cerr << "Cannot bind to port 15678."
                    << endl;
25             return -3;
26       }
27       if (listen(server_socket, 0) == -1)
28       {
29             cerr << "Cannot start listening"
                    << endl;
30             return -4;
31       }
32       int client_socket =
                      accept(server_socket, 0, 0);
33       if (client_socket == -1)
34       {
35             cerr << "Cannot obtain client
                       socket" << endl;
36             return -5;
37       }
38       cout << "Connection accepted" << endl;
39       char buf[100];
40       strcpy(buf, "Hello, Client!");
41       if (send(client_socket, buf,strlen(buf),0)
                                              == -1)
42       {
43             cerr << "Cannot send data via client
                       socket" << endl;
44             return -6;
```

```
45        }
46        if (recv(client_socket, buf, 100, 0)
                                              == -1)
47        {
48              cerr << "Cannot receive data via
                         client socket" << endl;
49              return -7;
50        }
51        cout << "Received data: " << buf << endl;
52        shutdown(client_socket, SHUT_RDWR);
53        close(client_socket);
54
55        shutdown(server_socket, SHUT_RDWR);
56        close(server_socket);
57        return 0;
58  }
```

Листинг 2.14: Пример клиента

```
1   #include <iostream>
2   #include <sys/socket.h>
3   #include <netinet/in.h>
4   #include <string.h>
5   #include <arpa/inet.h>
6
7   using namespace std;
8
9   int main()
10  {
11        int client_socket = socket(AF_INET,
                                       SOCK_STREAM, 0);
12        if (client_socket == -1)
13        {
14              cerr << "Cannot create socket"
                     << endl;
15              return -1;
16        }
17        sockaddr_in addr;
18        memset(&addr, 0, sizeof(sockaddr_in));
19        addr.sin_family = AF_INET;
20        addr.sin_addr.s_addr =
                              inet_addr("127.0.0.1");
21        addr.sin_port = htons(15678);
22        if (connect(client_socket,
        (sockaddr*)&addr, sizeof(sockaddr_in)) == -1)
```

```
23        {
24              cerr << "Cannot connect to
                          127.0.0.1:15678" << endl;
25              return -3;
26        }
27        char buf[100];
28        if (recv(client_socket, buf, 100, 0)== -1)
29        {
30              cerr << "Cannot receive data from
                         server" << endl;
31              return -4;
32        }
33        cout << "Received data: " << buf << endl;
34        strcpy(buf, "Hello, Server!!");
35        if (send(client_socket, buf, strlen(buf),
                                            0) == -1)
36        {
37              cerr << "Cannot send data to server"
                     << endl;
38              return -5;
39        }
40        shutdown(client_socket, SHUT_RDWR);
41        close(client_socket);
42        return 0;
43 }
```

2.3.4. Семафоры

При синхронизации процессов в среде Linux используются семафоры. Они представляются типом данных `int`. Для использования функций, связанных с семафорами, для начала нужно включить заголовочные файлы `<sys/ipc.h>` и `<sys/sem.h>`. Семафоры, однако, существуют лишь в множествах. То есть когда вы хотите создать один семафор, то на самом деле должны создать множество семафоров, в котором содержится лишь один семафор. Создать семафор можно при помощи функции `semget()`.

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

Нужно задать идентификатор множества семафоров, желаемое количество и флаги создания. Удаление семафора делается вызовом функции `semctl()` с аргументом `IPC_RMID`.

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

Операции увеличения и уменьшения значений семафоров реализуются вызовом функции `semop()`:

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

Смысл перечисленных функций станет более понятным после рассмотрения следующего примера.

Листинг 2.15: Синхронизация процессов

```
1  #include <iostream>
2  #include <sys/sem.h>
3  #include <sys/ipc.h>
4  #include <sys/types.h>
5
6  using namespace std;
7
8  union semun {
9     int val;
10    struct semid_ds *buf;
11    unsigned short int *array;
12    struct seminfo *__buf;
13 };
14
15 class Semaphore
16 {
17       public:
18             Semaphore(int Value);
19             void Up();
20             void Down();
21             ~Semaphore();
22
23       private:
24             int _SemID;
25 };
26
27 Semaphore::Semaphore(int Value)
28 {
29       _SemID = semget(IPC_PRIVATE, 1,
                         IPC_CREAT);
30
31       semun Init;
32       Init.val = Value;
33
```

```
34        semctl(_SemID, 1, SETVAL, Init);
35 }
36
37 void Semaphore::Up()
38 {
39        sembuf  ops[1];
40        ops[0].sem_num = 0; // Use the only
                                 semaphore in the set
41        ops[0].sem_op = 1; // Increment value by 1
42        ops[0].sem_flg = SEM_UNDO;
             // Undo in case if process is terminated
43        semop(_SemID, ops, 1);
44 }
45
46 void Semaphore::Down()
47 {
48        sembuf  ops[1];
49        ops[0].sem_num = 0;
             // Use the only semaphore in the set
50        ops[0].sem_op = -1; //Decrement value by 1
51        ops[0].sem_flg = SEM_UNDO;
             // Undo in case if process is terminated
52        semop(_SemID, ops, 1);
53 }
54
55 Semaphore::~Semaphore()
56 {
57        semun unused;
58        semctl(_SemID, 1, IPC_RMID, unused);
59 }
60
61 int main()
62 {
63        Semaphore sem(1);
64
65        if (fork() == 0) // Child process
66        {
67              whie (true)
68              {
69                    sem.Down();
70                    // Write somethign to some
                          shared memory object
71                    sem.Up();
72              }
73        }
```

```
74        else
75        {
76              cout << "Parent" << endl;
77              while (true)
78              {
79                    sem.Down();
80                    // Read the shared memory
                          contents and print it
81                    sem.Up();
82              }
83        }
84        return 0;
85  }
```

Как видно из листинга, нам пришлось вручную объявить новый тип для того, чтобы стало возможным использование семафоров. Обычно складывается ощущение, что функции, работающие с семафорами, не очень удобны для использования. Поэтому рекомендуется при использовании семафоров создать класс-оболочку подобно тому, как приведено в листинге 2.15.

2.4. Учебный пример: многоклиентный сервер вещания

В этой части пособия рассмотрим комплексный пример. Цель состоит в том, чтобы использовать изученные объекты и системные вызовы вместе в одном проекте. Также попытаемся шаг за шагом построить архитектуру приложения, что является очень полезным для разработчика приложений в любой среде.

2.4.1. Постановка задачи

Задачу можно сформулировать так: написать приложение, имеющее архитектуру «клиент-сервер», где сервер может одновременно обслуживать несколько клиентов. Сервер должен реализовать следующую функциональность: при получении данных от одного клиента он рассылает эти данные всем остальным клиентам. Читатель в функциональности сервера может найти схожесть с известным всем чат-сервером. На самом деле если данные будут строковыми, имеющими определенный формат, то данное приложение станет именно чатом.

2.4.2. Разработка архитектуры

Очевидно, что приложение состоит из двух относительно независимых частей: из сервера и клиента. Однако они зависимы в плане интерфейса. В данном случае интерфейс — это протокол (набор правил), по которому сервер и клиент будут обмениваться данными. Кроме этого, они имеют еще и общие части реализации. В частности, и сервер, и клиент должны иметь некую подсистему, отвечающую за передачу и получение данных. Назовем это транспортом. Лучше всего реализовать его как класс, экземпляры которого будут использованы в обеих частях. Далее общим также является целевой тип данных, представляющий передаваемые данные. Это может быть структура, которая фактически и задает протокол общения между сервером и клиентом.

Кроме перечисленного, и сервер, и клиент должны быть многопоточными. Следовательно, для удобства полезно создать класс-оболочку и для потоков. Что касается синхронизации, то разделяемые данные (скорее всего, это будет сообщение, полученное по сети) могут быть реализованы как монитор (специальный объект синхронизации, который инкапсулирует данные, предоставляя к ним только эксклюзивный доступ).

2.4.3. Детали реализации сервера

Итак, как говорилось выше, сервер — это многопоточная программа. В ней должно быть запущено несколько типов потоков. Во-первых, главный поток, который создает все необходимые объекты, инициализирует и ожидает соединения. При получении соединения главный поток запускает новый клиентский поток, обслуживающий клиента. Кроме этого, каждый клиентский сокет должен быть размещен в некотором разделяемом ресурсе (глобальной переменной). Это необходимо для потока вещания, запускаемого главным потоком. Функциональность потока вещания заключается в ожидании сообщений, и при получении рассылки их по всем клиентам. Тут и возникает проблема, которая решается реализацией синхронизации между клиентским и вещательным потоками.

2.4.4. Детали реализации клиента

Клиентское приложение является более простым. Тут должно быть запущено два потока: главный поток и поток, который отвечает за получение сообщений. На самом деле тут даже не возникает проблема синхронизации. Первый поток читает ввод из стандартного потока ввода, на базе этих данных он создает объект сообщения (согласно протоколу) и отсылает этот пакет серверу. Второй поток все время ожидает прибытия сообщений. При получении содержимое сообщения выводит в стандартный поток вывода.

Весь исходный код учебного примера может быть найден в архиве примеров в каталоге `Sources/chapter2/example` (см. раздел 6).

3. СОЗДАНИЕ ГРАФИЧЕСКИХ ПРИЛОЖЕНИЙ: GTK

В этой главе речь пойдет об основах разработки графических приложений в среде Linux. В Linux можно выделить две ведущие технологии разработки графических интерфейсов — это набор библиотек GNOME Development и набор библиотек Qt. В данной главе рассматривается первая технология, а Qt посвящена следующая глава.

В набор GNOME Development входят такие компоненты, как GTK+, Pango, Cairo, предназначенные непосредственно для создания графического пользовательского интерфейса, а также вспомогательные библиотеки, реализующие базовые операции и с графикой непосредственно не связанные: GLib, GObject и др.

Стоит заметить, что приложения, разработанные с использованием средств GNOME Development, как правило, могут работать не только в desktop-среде GNOME, но и в других, таких как, например, KDE.

3.1. Вспомогательные библиотеки

Чтобы написать полноценное графическое приложение с использованием средств GNOME, разработчикам нужно быть знакомыми не только со средствами вывода изображения на экран, но и со вспомогательными библиотеками, такими как GLib, GObject и др. Это библиотеки, часть которых напрямую не связана с выводом графики или пользовательских элементов интерфейса на экран, но без которых программисту пришлось бы самому снова «изобретать велосипед» для типовых операций. Это библиотеки, которые определяют полезные типы данных, интерфейсы для манипуляции экземплярами этих типов, утилитарные функции.

3.1.1. Библиотека GObject

Одной из основных среди вспомогательных библиотек является GObject. В ней определена иерархия типов данных, которая так или иначе используется всеми остальными библиотеками. В библиотеке GObject определен одноименный тип данных `GObject`, который является «предком» для всех остальных типов. Хотелось бы заметить, что хотя для всех библиотек существуют версии на языке C++ (классы-

оболочки), однако все они базируются на версиях библиотек, оригинально написанных на языке C. Следовательно, здесь не идет речь о полноценном объектно-ориентированном программировании и иерархии типов в смысле классов, здесь нет поддержки виртуальных функций и полиморфизма. Но разработчики GObject сделали почти невозможное, средствами языка C успешно реализуя псевдообъектно-ориентированную библиотеку. В библиотеке GObject, кроме известных членов структур (members), каждый тип обладает еще двумя видами атрибутов — *свойствами* (properties) и *сигналами* (signals).

Свойства типов — это именованные поля, доступ к которым может быть ограничен. Бывают как свойства только для чтения, так и только для записи. Есть свойства, определяемые при создании объекта и недоступные ни для чтения, ни для записи. Эти ограничения доступа позволяют создать довольно гибкую систему атрибутов объекта.

Сигналы — это сообщения, которые могут быть отправлены объектам, заинтересованным в том или ином изменении в состоянии объекта. Например, при нажатии на кнопку генерируется сигнал `clicked`, рассылаемый всем заинтересованным объектам, которые должны отреагировать на это событие. Именно на сигналах объектов и базируется программирование на основе событий (Event-based programming) в среде GNOME Development.

3.1.2. Библиотека GLib

Библиотека GLib содержит большое количество переопределенных базовых и новых типов, а также различные функции-манипуляторы над объектами этих типов. В библиотеку входят такие типы, как `gchar`, `gint`, `guint`, `gpointer` и т.д. Например, `gpointer` — это указатель без типа, то есть это аналог типа `void*`. Для всех остальных примитивных типов `GLib` также есть аналоги в системе базовых типов языка C. Кроме примитивных типов, библиотека `GLib` предоставляет очень широкое множество составных типов, таких как строки, связанные списки, деревья, хеш-таблицы и т.д. Для всех этих типов определены все основные операции, что может очень сильно облегчить процесс разработки приложений.

Кроме перечисленных возможностей, библиотека GLib предоставляет оболочки для нескольких системных объектов-понятий. Например, GLib предоставляет собственную версию потоков, которая отличается от стандартных потоков POSIX (см. 2.2). В GLib также существуют собственные реализации каналов ввода-вывода, выделе-

ния памяти, динамической загрузки модулей, системы ошибок, записи журнала и т.д. Однако в этом множестве возможностей следует особенно выделить *механизм основного цикла приложения* (main loop). Как известно, каждое приложение, которое создано с использованием событийного программирования, представляет собой один большой цикл, в котором обрабатываются сообщения. Приложения в среде GNOME Development реализуют этот цикл именно средствами библиотеки GLib (или же GTK+, которое, на самом деле, предоставляет более удобную оболочку).

Поскольку описание всех возможностей библиотеки GLib заняло бы огромный объем, читателю рекомендуется ознакомиться с деталями самостоятельно. Очень удобным инструментом для этого является утилита `devhelp`. В ней собраны руководства пользователей всех основных библиотек.

Тем не менее, стоит упомянуть наиболее популярные библиотеки, чтобы читатель имел представление об их использовании. В листинге 3.1 приведена программа, которая открывает файл (с известной заранее кодировкой), конвертирует его содержимое при помощи нескольких потоков и записывает результат в другой файл. Для определенности предположим, что исходный файл имеет кодировку ISO8859-1, а результат должен иметь кодировку UTF8.

Листинг 3.1: Полноценная программа на GLib

```
1  #include <iostream>
2  #include <glib.h>
3  using namespace std;
4
5  const gchar * SourceFile = "source.txt";
6  const gchar * DestFile = "dest.txt";
7  const gchar * SourceEnc = "ISO-8859-1";
8  const gchar * DestEnc = "UTF8";
9
10 const int ThreadCount = 4;
11
12 struct ThreadParam
13 {
14       gchar * Source;
15       gsize SourceLength;
16       gchar * Result;
17       gsize ResultLength;
18 };
```

```
19
20 gpointer ThreadProc(gpointer param)
21 {
22       ThreadParam * data = (ThreadParam*)param;
23
24       data->Result = g_convert(data->Source,
             data->SourceLength, DestEnc, SourceEnc,
             NULL, &data->ResultLength, NULL);
25
26       return data;
27 }
28
29 int main(int argc, char ** argv)
30 {
31       g_thread_init(NULL);
32
33       gsize SourceLength;
34       gchar * SourceContents;
35       if ( g_file_get_contents(SourceFile,
                &SourceContents, &SourceLength, NULL)
             == FALSE)
36       {
37             cerr << "Cannot open source file "
                       << SourceFile << endl;
38             return -1;
39       }
40
41       GThread * Threads[ThreadCount];
42       ThreadParam * Params =
                      new ThreadParam[ThreadCount];
43
44       for (int i = 0; i < ThreadCount; ++i)
45       {
46             ThreadParam * data = &Params[i];
47             data->Source = SourceContents +
                       i * SourceLength / ThreadCount;
48             data->SourceLength =
               ((i + 1) * SourceLength / ThreadCount
                > SourceLength)
               ? (SourceLength - (i + 1) *
                  SourceLength / ThreadCount + 1)
               : SourceLength / ThreadCount + 1;
49             Threads[i] = g_thread_create(
                        GThreadFunc(ThreadProc),
```

```
                        (gpointer)data,
                        TRUE, NULL);
50      }
51
52      for (int i = 0; i < ThreadCount; ++i)
53      {
54            g_thread_join(Threads[i]);
55      }
56
57      gsize ResultLength = 0;
58      gchar ** ConvertedStrings =
                        new gchar*[ThreadCount + 1];
59      for (int i = 0; i < ThreadCount; ++i)
60      {
61            ResultLength +=
                              Params[i].ResultLength;
62            ConvertedStrings[i] =
                              Params[i].Result;

63      }
64      ConvertedStrings[ThreadCount] = NULL;
65
66      gchar * ResultContents = g_strjoinv("",
                                    ConvertedStrings);
67      for (int i = 0; i < ThreadCount; ++i)
68      {
69            g_free(Params[i].Result);
70      }
71      g_free(SourceContents);
72
73      if (g_file_set_contents
              (DestFile, ResultContents,
               ResultLength, NULL) == FALSE)
74      {
75            cerr << "Cannot save result to "
                       << DestFile << endl;
76            return -2;
77      }
78      cout << "Output written to " << DestFile
              << endl;
79      g_free(ResultContents);
80      return 0;
81 }
```

Проведем более детальный анализ приведенного кода. Отметим, что в этом приложении использовано несколько подсистем библиотеки GLib: `GThread`, `GFile`, `GConvert`, `GString`. Перед использованием функциональности потоков необходимо инициализировать подсистему потоков. Это делается при помощи функции `g_thread_init()`. Для чтения содержимого исходного файла используется функция `g_file_get_contents()`:

```
gboolean g_file_get_contents(const gchar *filename,
                             gchar ** contents,
                             gsize *length,
                             GError ** error);
```

Она принимает как входной параметр имя файла, содержимое которого нужно прочесть. Остальные же параметры являются выходными. В параметре `contents` возвращается содержимое файла, а в параметре `length` — его длина. Если в качестве параметра `error` передать ненулевой указатель, то в него будут записаны ошибки, которые выявились при чтении файла. При успешном чтении функция возвращает `TRUE`, в случае же неудачи — `FALSE`.

Далее идет подготовка к запуску потоков. Во-первых, создается массив `GThread*`, где будут содержаться объекты потоков. Также нам необходим массив `ThreadParam` для хранения параметров потоков. Структура `ThreadParam` содержит собственно саму строку для конвертации, ее длину и выходные поля, в которые записывается результат конвертации и его длина. Полагается, что каждый из рабочих потоков проводит конвертацию и заполняет вышеупомянутые поля переданного ему параметра. Содержимое исходного файла по возможности равномерно разделяется между несколькими потоками (их количество задает константа `ThreadCount`). Запуск потоков производится вызовом функции `g_thread_create()`:

```
GThread * g_thread_create (GThreadFunc func,
                           gpointer data,
                           gboolean joinable,
                           GError ** error);
```

Параметр `func` — это указатель на поточную функцию со следующим прототипом:

```
gpointer (*GThreadFunc) (gpointer data);
```

Параметр `data` — это данные, передающиеся поточной функции при ее выполнении. Тип этого параметра `gpointer` представляет собой аналог типа `void*`, это универсальный указатель. Параметр `joinable` указывает на то, поддерживает создаваемый поток ожидание или нет. Если в качестве третьего параметра передать значение `TRUE`, то создающий поток далее может подождать, пока вновь созданный поток не завершит свое выполнение. В нашем случае требуется именно такая функциональность. Ожидание завершения рабочего потока производится вызовом функции `g_thread_join()`:

```
gpointer g_thread_join (GThread *thread);
```

Функция возвращает значение, возвращаемое поточной функцией. После того как все потоки завершили свое выполнение, основной поток собирает полученные значения в один массив (массив `ConvertedStrings`), который далее обрабатывается при помощи функции `g_strjoinv()`:

```
gchar* g_strjoinv (const gchar * separator,
                   gchar ** str_array);
```

Эта функция создает новую строку из фрагментов, задаваемых параметром `str_array`. Между каждыми двумя соседними фрагментами вставляется строка, заданная в параметре `separator`. В нашем примере сепаратор — это пустая строка.

3.1.3. Библиотека Pango

До сих пор мы не обращали внимания на имена библиотек GNOME. Имена GObject и GLib достаточно понятны. Но дело обстоит иначе с библиотекой Pango. Ее название состоит из двух частей — `Pan` и `Go`. Эти два слова взяты из двух разных языков. `Pan` — греческое слово, которое означает «все». `Go` — по-японски означает «язык». Следовательно, Pango на комбинированном языке означает «все языки». Pango — это библиотека, предназначенная для работы со строками и шрифтами. Из названия становится понятно, что эта библиотека поддерживает все языки мира. К библиотеке Pango приходится обращаться всегда, когда дело доходит до вывода текста на экран.

Одной из самых удивительных возможностей Pango является поддержка языка разметки The Pango Markup Language. Это язык разметки текста, имеющий некоторое сходство с языком HTML. Например, строка

```
<b>This text is bold!</b>
```

после обработки Pango будет выглядеть так:

This text is bold!

Поддерживается не очень большое подмножество тэгов HTML, которое, однако, достаточно для базового форматирования текста. В число поддерживаемых тэгов входят `<b>`, `<i>`, `<s>`, `<sup>`, `<sub>` и, конечно же, `<span>` с различными атрибутами (поддерживаются `font family`, `size`, `weight`, `foreground`, `background`, `lang` и другие).

Поскольку у Pango такое особое назначение, то проблематично привести пример приложения, где используется только она. Пример использования Pango будет приведен позднее в сочетании с GTK+ (см.: листинг 3.7).

3.1.4. Библиотека GDK

Библиотека GDK (GTK+ Drawing Kit), как следует из названия, является основной подсистемой, отвечающей за рисование для GTK+. Точнее сказать, все, что выводится на экран, кроме текста, выводится посредством библиотеки GDK. Для наглядности можно заметить, что основной тип GTK+ — `GtkWidget` — это структура, содержащая один указатель на тип `GdkWindow`. Иными словами, все объекты GTK+ для визуализации используют объект типа `GdkWindow` из библиотеки GDK. `GdkWindow` — это прямоугольный участок на экране, где можно что-нибудь отображать. Он может иметь или не иметь заголовок, рамку и т.д.

Из большого множества функций библиотеки GDK разработчикам для непосредственного использования требуется знать лишь небольшую часть, так как большинство функций GDK используется через GTK+. Стоит изучить функции, предназначенные для рисования базовых графических объектов, наряду с соответствующими вспомогательными объектами.

Все интерфейсные функции библиотеки GDK начинаются префиксом `gdk_`. За префиксом следует имя объекта, к которому относится данная функция, только после этого идет имя операции. Например, функция `gdk_gc_set_values` относится к объекту `GdkGC` (Graphics Context) и, как следует из имени, с ее помощью можно задать значения для графического контекста.

В листинге 3.2 показан исходный код программы, использующей GDK.

Листинг 3.2: Первая программа на GDK

```
1  #include <unistd.h>
2  #include <gdk/gdk.h>
3
4  int main(int argc, char** argv)
5  {
6        gdk_init(&argc, &argv);
7        GdkWindowAttr attrs;
8        attrs.title = (gchar*)"Demo window";
9        attrs.width = 800;
10       attrs.height = 600;
11       attrs.x = 100;
12       attrs.y = 100;
13       attrs.window_type = GDK_WINDOW_TOPLEVEL;
14       GdkWindow * window =
              gdk_window_new(NULL, &attrs,
               GDK_WA_TITLE | GDK_WA_X | GDK_WA_Y);
15       gdk_window_show(window);
16       gdk_flush();
17       sleep(2);
18       return 0;
19 }
```

Читателю может показаться странной строка 17, где вызывается функция `sleep()`. Дело в том, что поскольку тут нет цикла-обработчика, то программа завершит свое выполнение, так и не показывая окно. Вызов `sleep()` позволяет отложить завершение программы, тем самым позволяя увидеть созданное окно. Еще одной уловкой является вызов `gdk_flush()`. Эта функция заставляет GDK непременно выполнять все предыдущие операции вместо буферизации. В частности, без вызова `gdk_flush()` окно программы не показывалось бы.

Разумеется, это просто демонстрационный пример. Необходимость разрабатывать приложения с использованием только низкоуровневых библиотек GDK без GTK на практике возникает довольно редко. Даже приведенные здесь функции `gdk_window_new()` и `gdk_window_show()` тоже вряд ли придется использовать. Однако, как уже говорилось, есть определенное подмножество типов и функ-

ций из GDK, которые придется использовать наряду с GTK+. В листинге 3.3 приведен исходный код полноценной программы без использования GTK+. В результате выполнения данного примера в окне приложения будут нарисованы сплошной красный квадрат, квадрат с пунктирной зеленой границей и пунктирная синяя окружность.

Листинг 3.3: Полноценная программа на GDK

```
1  #include <unistd.h>
2  #include <gdk/gdk.h>
3  #include <cairo/cairo.h>
4  #include <iostream>
5
6  using namespace std;
7
8  GdkWindow * window;
9  GMainLoop * loop;
10
11 void event_handler(GdkEvent* event,
                       gpointer data)
12 {
13       cout << "Event: " << event->type << endl;
14       if ( event->type == GDK_EXPOSE )
15       {
16             cout << "\tExpose" << endl;
17
18             cairo_t *cr;
19             cr = gdk_cairo_create(window);
20
21             cairo_set_source_rgb(cr, 1, 0, 0);
22             cairo_rectangle(cr, 10,10,100,100);
23             cairo_fill(cr);
24
25             double dashes[2] = {10, 10};
26             cairo_set_line_cap(cr,
                            CAIRO_LINE_CAP_BUTT);
27             cairo_set_line_join(cr,
                            CAIRO_LINE_JOIN_ROUND);
28             cairo_set_dash(cr, dashes, 2, 0);
29             cairo_set_source_rgb(cr, 0, 1, 0);
30             cairo_rectangle(cr, 10,115,100,100);
31             cairo_stroke(cr);
32
33             cairo_set_source_rgb(cr, 0, 0, 1);
```

```
34              cairo_arc(cr, 60, 275, 50, 0, 360);
35              cairo_stroke(cr);
36
37              cairo_destroy(cr);
38
39              return;
40        }
41        if (event->type == GDK_DESTROY ||
               event->type == GDK_DELETE)
42        {
43              cout << "\tDestroying" << endl;
44              gdk_window_destroy(window);
45              g_main_loop_quit(loop);
46        }
47 }
48
49 int main(int argc, char** argv)
50 {
51        gdk_init(&argc, &argv);
52        GdkWindowAttr attrs;
53        attrs.title = (gchar*)"Drawing window";
54        attrs.width = 800;
55        attrs.height = 600;
56        attrs.x = 100;
57        attrs.y = 100;
58        attrs.wclass = GDK_INPUT_OUTPUT;
59        attrs.event_mask = GDK_EXPOSURE_MASK;
60        attrs.window_type = GDK_WINDOW_TOPLEVEL;
61
62        window = gdk_window_new(NULL, &attrs,
                GDK_WA_TITLE | GDK_WA_X | GDK_WA_Y );
63        if (!window)
64        {
65              return -1;
66        }
67
68        gdk_event_handler_set(event_handler,
                                  NULL, NULL);
69        gdk_window_show(window);
70
71        GdkRectangle rect;
72        gdk_window_get_size(window, &rect.width,
                                     &rect.height);
73        gdk_window_invalidate_rect(window,
                                     &rect, FALSE);
```

```
74        loop = g_main_loop_new(NULL, TRUE);
75        g_main_loop_run(loop);
76        return 0;
77 }
```

Рассмотрим подробнее ключевые части приведенного кода. Первая вызванная функция — это `gdk_init()`. Без ее вызова ни одна программа на GDK не может выполняться. Далее идет вызов функции `gdk_event_handler_set()`. С ее помощью задается функция (в нашем случае `event_handler()`), которая должна вызываться при получении любых событий. Далее создается основное окно программы. Особый интерес представляют строки 74 и 75. В строке 74 создается объект типа `GMainLoop`, активизируемый в строке 75. Именно этот объект представляет собой основной цикл программы и делает возможным вызов обработчика сообщений.

Замечание 3.1.1. В этом примере использована библиотека Cairo, так как собственные средства вывода графических объектов в библиотеке GDK устарели (deprecated).

Рассмотрим поближе код обработчика сообщений. Это функция с определенным прототипом:

```
void (*GdkEventFunc) (GdkEvent *, gpointer);
```

Эта функция, как и все обработчики сообщений, независимо от платформы, представляет собой набор ветвлений (switch по событиям). Тут должны отражаться все те события, на которые приложение должно реагировать. В нашем случае обрабатываются сообщения о перерисовке (`GDK_EXPOSE`) и о разрушении или закрытии окна (`GDK_DESTROY` и `GDK_DELETE`). Поскольку обо всех остальных событиях ничего не говорится, то они будут просто игнорированы приложением. Разумеется, что в приложениях, которые написаны с использованием GTK+, такого обработчика не будет. Вместо этого будут конкретные функции для обработки конкретных сообщений.

3.2. Библиотека GTK+

После начального знакомства со вспомогательными библиотеками GNOME пора перейти к библиотеке GTK+, о которой уже не раз говорилось. Начнем с имени библиотеки. В полностью развернутом виде оно будет следующим: Gnu's Not Unix Network Object Model Environment Image Manipulation Program ToolKit. Оказывается, что GTK+ изначально являлась вспомогательной библиотекой для разработчиков одного приложения — GIMP — GNOME Image Manipulation Program. Но со временем и с ростом функциональности библиотеки она приобрела иное, более важное значение. Значительная часть созданного к настоящему времени открытого программного обеспечения использует GTK+.

3.2.1. Структура приложения

Приложения, которые используют GTK+, должны иметь определенную структуру. Эта необходимость исходит от того, что для правильного использования данной библиотеки нужно проделать некоторые шаги в определенной последовательности. Для начала обязательно вызывается функция `gtk_init()`:

```
void gtk_init (int *argc, char *** argv);
```

Эта функция выполняет инициализацию компонентов GTK. В частности, инициализирует GUI. При неудаче она завершает работу приложения. Кроме того, данной функции можно передать параметры из командной строки для дополнительных настроек.

После успешной инициализации идет определение и инициализация переменных, а также определение обработчиков сигналов. Как говорилось выше, все типы из `GObject` (а также и GTK+) снабжены сигналами. Они генерируются в ответ на различные изменения. Приложение должно регистрировать обработчики для тех сигналов, которые его интересуют. Для привязки обработчика к сигналу можно использовать GTK-функцию `gtk_signal_connect()`. Однако эта функция объявлена устаревшей и не рекомендуется к дальнейшему использованию. Вместо нее нужно использовать функцию `g_signal_connect()`:

```
gulong g_signal_connect (gpointer instance,
                         const gchar * detailed_signal,
                         GCallback c_handler,
                         gpointer gobject);
```

Примечание 3.2.1. На самом деле функции с таким прототипом не существует. Но есть макрос с этим именем, аналогичный приведенной функции.

Первый параметр — это указатель на объект, сигнал которого мы хотим обрабатывать. Второй параметр — это строковое значение, содержащее имя сигнала. Третий параметр — это указатель на функцию обработчика. Четвертый параметр — это указатель, который передается обработчику при вызове.

После того, как все обработчики определены, нужно отобразить основное окно приложения вместе со всеми элементами в нем. Отображать объект можно вызовом функции `gtk_widget_show()`:

```
void gtk_widget_show (GtkWidget *widget);
```

Примечание 3.2.2. Тип `GtkWidget` является базовым типом для всех остальных объектов (окна, кнопки, спадающие списки и т.д.).

Эта функция отображает сам указанный виджет, игнорируя объекты, находящиеся на нем. Для того чтобы отображалось все, необходимо вызвать функцию `gtk_widget_show_all()`:

```
void gtk_widget_show_all (GtkWidget *widget);
```

Более подробно об основных типах и о функциях можно узнать в 3.2.3.

Когда основное окно приложения отображено, нужно активировать основной цикл обработки сообщений. В GTK+ это делается с помощью функции `gtk_main()`. Данная функция не принимает никаких параметров, а завершает выполнение при вызове функции `gtk_main_quit()`. Эти функции на самом деле являются оболочками для функциональности, о которой шла речь в 3.1.2 (см. также листинг 3.3). В листингах, приведенных в 3.2.2, показано, как именно должно быть организовано приложение GTK+.

3.2.2. Первая программа на GTK+

По традиции в качестве первого примера напишем программу, которая выводит выражение *Hello, World!*. В листинге 3.4 приведен исходный код такого приложения.

Листинг 3.4: Hello World! на GTK+

```
1  #include <gtk/gtk.h>
2
3  GtkWidget * main_window;
4
5  void Redraw()
6  {
7        PangoLayout * layout =
                   gtk_widget_create_pango_layout
                     (main_window, "Hello, World!");
8        GdkGC * gc = gdk_gc_new
                              (main_window->window);
9        GdkColor color;
10       gdk_color_parse("#FF0000", &color);
11       gdk_gc_set_rgb_fg_color(gc, &color);
12       gdk_draw_layout(main_window->window, gc,
                                  110, 60, layout);
13 }
14
15 int main(int argc, char ** argv)
16 {
17       gtk_init(&argc, &argv);
18       main_window =
               gtk_window_new(GTK_WINDOW_TOPLEVEL);
19       gtk_window_set_title
         (GTK_WINDOW(main_window), "Hello, World!");
20       gtk_window_resize(GTK_WINDOW(main_window),
                                        300, 150);
21       gtk_widget_show_all(main_window);
22       g_signal_connect(main_window,
                     "expose-event", Redraw, NULL);
23       gtk_main();
24       return 0;
25 }
```

Читателю может показаться странным то, что минимальная программа содержит 25 строк, однако, если написать аналогичную про-

грамму с использованием Win32 API, то придется написать куда больше строк кода.

3.2.3. Основные типы в GTK+

Переходя к основным типам библиотеки GTK+, следует напомнить читателю, что эта библиотека имеет «псевдообъектно-ориентированный» стиль, несмотря на то, что она написана на языке C. Объектно-ориентированная часть библиотеки поддерживается средствами библиотеки `GObject`. То есть все объекты (как логические понятия в отличие от объектов, присущих настоящим объектно-ориентированным языкам), помимо стандартных членов-данных, имеют также свойства и сигналы, передаваемые «по наследству».

При использовании GTK+ на начальном уровне самым важным типом является тип `GtkWidget`. Он является базовым типом для всех типов, которые представляют объекты пользовательского интерфейса: `GtkWindow`, `GtkButton`, `GtkMenu` и т.д. Почти все функции, инициализирующие объекты, возвращают объект типа `GtkWidget*`. С одной стороны, это приводит к некоторой универсальности. С другой стороны, такой подход приводит к некоторым неудобствам. Все функции, предназначенные для манипуляции какими-либо конкретными объектами, в качестве первого параметра принимают указатель именно на данный тип. Например, функция `gtk_window_set_title()` принимает указатель на `GtkWindow`. А поскольку при создании окна использовался тип `GtkWidget*`, то его нужно привести к типу `GtkWindow*` (что и делается в листинге 3.4 в строке 19). Для приведения типов в библиотеке GTK+ предусмотрен ряд макросов. Все они начинаются с префикса GTK , а дальше идет имя типа заглавными буквами. Если данное приведение невозможно, то GTK+ выдаст предупреждение или ошибку на консоль.

Тип `GtkWidget` содержит все свойства и сигналы, которые являются общими для всех типов-наследников. Например, именно в `GtkWidget` определены следующие свойства: `name`, `parent`, `style`, `visible` и др. Очень важным является свойство `window`. Оно также доступно как член структуры, имеет тип `GdkWindow*` и представляет собой объект GDK, лежащий в основе всех объектов пользовательского интерфейса. Именно при помощи этого свойства становится доступным графическая подсистема GDK. Ведь тип `GdkWindow*`

можно привести к типу GdkDrawable*, который используется всеми функциями рисования в библиотеке GDK.

Из большого множества сигналов, поддерживаемых типом GtkWidget, можно выделить следующие: button-press-event, button-release-event, key-press-event, key-release-event, motion-notif-event и др. Эти сигналы представляют соответственно события нажатия и отпускания кнопки мыши и клавиатуры и движения мыши в области окна. Использованный ранее expose-event тоже является сигналом GtkWidget.

Именно при помощи типа GtkWidget становится возможным отображение окон и остальных элементов пользовательского интерфейса. Делается это при помощи функции gtk_widget_show().

```
void gtk_widget_show (GtkWidget *widget);
```

Эта функция отображает объект widget на дисплее. Однако она не очень удобна в силу следующих соображений. В GTK+ некоторые объекты представляют собой контейнеры. Некоторые из них имеют ограничения на множество дочерних объектов, а другие таких ограничений не имеют. Следовательно, пользовательский интерфейс представляет собой множество объектов, которые каким-то образом вложены друг в друга. При вызове функции gtk_widget_show(widget) отображается лишь объект widget, а объекты, содержащиеся в нем, остаются невидимыми. Одним из вариантов отображения всех объектов является рекурсивный обход всех дочерних объектов и вызов функции gtk_widget_show() для каждого из них или же можно использовать функцию gtk_widget_show_all().

```
void gtk_widget_show_all (GtkWidget *widget);
```

Следующим по значимости типом в GTK+ можно считать тип GtkWindow, соответствующий окнам. Окно — это прямоугольная область на дисплее, которая может иметь границу, заголовок, системное меню, кнопки свертывания, максимизации, закрытия и другие атрибуты. Окно можно создать вызовом функции gtk_window_new().

```
GtkWidget* gtk_window_new (GtkWindowType type);
```

Параметр type может иметь одно из двух возможных значений: GTK_WINDOW_TOPLEVEL или GTK_WINDOW_POPUP, означающие соот-

ветственно основное и всплывающее окна. Возвращает она, как говорилось ранее, объект типа `GtkWidget*`.

Суммируя вышесказанное, можно рассмотреть пример, где используются сигналы, связанные с мышью. Это уже полноценная программа, которая будет закрываться вместе с основным окном.

Листинг 3.5: Полноценное приложение на GTK+: gtk_paint

```
1  #include <gtk/gtk.h>
2  #include <vector>
3  #include <algorithm>
4  #include <assert.h>
5  #include <iostream>
6  using namespace std;
7
8  vector<GdkPoint> Points;
9  GtkWidget * MainWindow = NULL;
10 GdkPoint CurrentPos;
11 gboolean DrawEndingLine = FALSE;
12 void InvalidateWindow(GtkWidget * window);
13
14 void Redraw()
15 {
16       assert(MainWindow != NULL);
17
18       if ( Points.size() == 0 )
19             return;
20
21       GdkColor color;
22       gdk_color_parse("#FFFF00", &color);
23       cairo_t * cr = gdk_cairo_create
                (GDK_DRAWABLE(MainWindow->window));
24
25       if ( !cr )
26             return;
27
28       gdk_cairo_set_source_color(cr, &color);
29       cairo_move_to(cr,
                     Points[0].x, Points[0].y);
30
31       for (unsigned int i = 1;
              i < Points.size(); ++i )
32       {
```

```
33              cairo_line_to(cr,
                          Points[i].x, Points[i].y);
34        }
35        cairo_line_to(cr,
                        CurrentPos.x, CurrentPos.y);
36        cairo_stroke(cr);
37        cairo_destroy(cr);
38
39  }
40
41  void ButtonReleaseHandler(GtkWidget * widget,
                              GdkEventButton * event)
42  {
43        DrawEndingLine = TRUE;
44        GdkPoint tmp;
45        tmp.x = event->x;
46        tmp.y = event->y;
47
48        Points.push_back(tmp);
49        InvalidateWindow(MainWindow);
50  }
51
52  void MouseMoveHandler(GtkWidget * widget,
                              GdkEventMotion * event)
53  {
54        CurrentPos.x = event->x;
55        CurrentPos.y = event->y;
56        if (DrawEndingLine)
57              InvalidateWindow(MainWindow);
58  }
59
60  void Destroy()
61  {
62        gtk_main_quit();
63  }
64
65  int main(int argc, char ** argv)
66  {
67        gtk_init(&argc, &argv);
68        MainWindow = gtk_window_new
                               (GTK_WINDOW_TOPLEVEL);
69        gtk_window_set_title
            (GTK_WINDOW(MainWindow), "Paint lines!");
70        gtk_window_resize(GTK_WINDOW(MainWindow),
                                          400, 300);
```

```
71       gtk_widget_set_events(MainWindow,
                               GDK_ALL_EVENTS_MASK);
72       g_signal_connect(MainWindow,
                     "expose-event", Redraw, NULL);
73       g_signal_connect(MainWindow, "destroy",
                                      Destroy, NULL);
74       g_signal_connect(MainWindow,
                           "button-release-event",
                    G_CALLBACK(ButtonReleaseHandler),
                    NULL);
75       g_signal_connect(MainWindow,
                           "motion-notify-event",
                    G_CALLBACK(MouseMoveHandler),
                    NULL);
76       gtk_widget_show_all(MainWindow);
77
78       gtk_main();
79       return 0;
80 }
81
82 void InvalidateWindow(GtkWidget * window)
83 {
84       GdkRectangle rect;
85       GdkPoint LastPoint =
                           Points[Points.size() - 1];
86       rect.x = min(LastPoint.x, CurrentPos.x)
                                                - 100;
87       rect.y = min(LastPoint.y, CurrentPos.y)
                                                - 100;
88       rect.width = abs(LastPoint.x-CurrentPos.x)
                                                + 100;
89.      rect.height =abs(LastPoint.y-CurrentPos.y)
                                                + 100;
90       gdk_window_invalidate_rect
                     (window->window, &rect, TRUE);
91 }
```

Внешний вид программы представлен на рисунке 3.1.

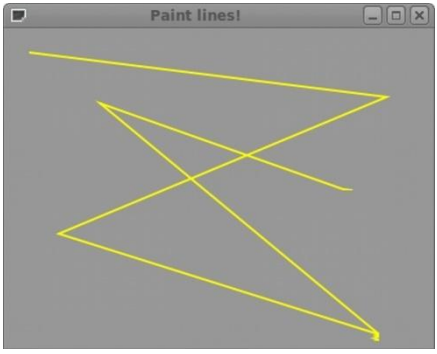

Рис. 3.1. Окно примера gtk_paint

Как и в прошлом примере, вся функциональность, связанная с перерисовкой содержимого окна, собрана в одну функцию — в обработчик сигнала `expose-event`. Кроме этого, приложение обрабатывает сигналы `button-release-event`, `motion-notify-event`. И особенно нужно подчеркнуть сигнал `destroy`, обработчик которого завершает основной цикл приложения. Точки, где пользователь нажимал любую из кнопок мыши, собираются в один вектор. Кроме этого, в глобальной переменной хранится текущая позиция мыши. При перерисовке функция `Redraw()` выводит ломаную с вершинами в вышеупомянутых точках. Особый интерес представляет функция `InvalidateWindow()`. В этой функции определяется участок окна для перерисовки и отображения последних изменений. Перерисовывается не все окно, а лишь прямоугольник, содержащий последние две точки.

Перед тем как перейти к остальным типам GTK+, хотелось бы отметить еще одно обстоятельство. В библиотеке GTK+, как и во многих других, имена функций построены по определённым, общим для всей библиотеки правилам. Все функции имеют префикс `gtk_`, после чего идет собственно имя объекта. И только после этого следует имя операции. Например, функция `gtk_button_set_label()`, судя по ее имени, должна определить текст, который должен быть на кнопке, что и соответствует действительности. Следовательно, иногда, даже не зная имени какой-либо функции, можно догадаться о ее назначении. Поскольку в начале книги мы условились использовать

Geany как среду разработки приложений, то следует знать, как ее можно настроить для удобного использования при создании приложений на GTK+. Перейдите в каталог `/.config/geany/tags` (если такой каталог не существует, то создайте его). В этом каталоге нужно запустить следующую команду:

```
CFLAGS=`pkg-config --cflags gtk+-2.0` geany –g
       gtk.cpp.tags  /usr/include/gtk-2.0/gtk/gtk.h
```

Эта команда запускает Geany в специальном режиме. Она создает файл тегов из файла `gtk.h`. После перезагрузки Geany станут доступными подсказки и автодополнения типов, макросов и функций GTK+.

Элементы пользовательского интерфейса можно разбить на группы в зависимости от их назначения, что очень хорошо и детально сделано в руководстве пользователя GTK+, которое можно найти на официальном сайте GTK+ (см. [5]). Поэтому в данном пособии об элементах пользовательского интерфейса будет сказано лишь поверхностно. Более важным является понимание того, как все эти элементы должны быть организованы и как происходит их взаимодействие, нежели перечень всех имен типов и функций.

Как говорилось ранее, некоторые из элементов пользовательского интерфейса представляют собой контейнеры. Контейнеры по их вместимости бывают двух типов: одноместные и многоместные. Например, `GtkWindow` — одноместный контейнер. Это означает, что в окне может содержаться лишь один объект. Следовательно, если нужно создать сложный интерфейс, то в самом окне следует поставить некоторый многоместный контейнер, в который уже можно будет разместить другие элементы интерфейса. В GTK+ существует целый ряд типов, представляющих разные контейнеры. В их число входят следующие типы: `GtkFixed`, `GtkTable`, `GtkNotebook`, `GtkLayout`, `GtkExpander`, `GtkHBox`, `GtkVBox` и другие.

Элементы в контейнер можно добавлять при помощи функции `gtk_container_add()`.

```
void gtk_container_add (GtkContainer * container,
                        GtkWidget * widget);
```

Эта функция добавляет `widget` к контейнеру `container`. Она удобна для примитивных контейнеров, таких как `GtkWindow`, `GtkFrame`. Однако для более сложных контейнеров лучше использо-

вать специальные функции. В листинге 3.6 показано, как можно добавить кнопку к основному окну приложения и как обрабатывается событие нажатия кнопки.

Листинг 3.6: Использование кнопки

```
1  #include <gtk/gtk.h>
2  #include <iostream>
3  using namespace std;
4
5  GtkWidget * MainWindow;
6
7  void ButtonClicked()
8  {
9        cout << "Button clicked" << endl;
10 }
11
12 int main(int argc, char ** argv)
13 {
14       gtk_init(&argc, &argv);
15       MainWindow =
                gtk_window_new(GTK_WINDOW_TOPLEVEL);
16       gtk_window_set_title
            (GTK_WINDOW(MainWindow), "Button test");
17       gtk_window_resize(GTK_WINDOW(MainWindow),
                                          300, 150);
18       GtkWidget * Button =
             gtk_button_new_with_label("Click me!");
19       gtk_container_add(GTK_CONTAINER
                                (MainWindow), Button);
20       gtk_widget_show_all(MainWindow);
21       g_signal_connect(Button, "clicked",
                                ButtonClicked, NULL);
22       gtk_main();
23       return 0;
24 }
```

Если скомпилировать и запустить приложение, то можно заметить, что кнопка заполняет все пространство в окне. Дело в том, что GTK+ автоматически распределяет объекты в контейнере так, чтобы использовать все пространство. Однако есть и контейнеры, для которых возможно определение координат содержащихся объектов. Одним из таких является `GtkFixed`. Определять расположение дочерних

объектов можно при помощи функций `gtk_fixed_put()` и `gtk_fixed_move()`.

Кроме сравнительно простых элементов интерфейса, как `GtkButton`, `GtkEntry`, `GtkLabel` и др., в GTK+ есть определенный класс типов, представляющих собой составные объекты-диалоги. Например, `GtkAboutDialog` представляет собой диалоговое окно, где отображается информация о приложении (автор, лицензия, и т.д.). Есть также несколько диалогов, которые предназначены для выбора некоторых величин. Например, тип `GtkColorSelectionDialog` позволяет пользователю выбрать цвет из очень удобной палитры. Также есть типы `GtkFileChooserDialog` для выбора файла (при открытии или сохранении) и `GtkFontSelectionDialog` — для выбора шрифта.

3.2.4. Сочетание GTK+ со вспомогательными библиотеками

Как уже видно из приведенных примеров, библиотека GTK+ часто используется в сочетании с одним или несколькими вспомогательными библиотеками. В частности, в листингах 3.5 и 3.2.2 показано взаимодействие GTK+ и GDK.

В качестве первого примера рассмотрим приложение, которое поддерживает выбор шрифта, размера, ориентации и цвета текста (примитивный «word-art»). В листинге 3.7 приведен исходный код примера.

Листинг 3.7: GTK+ и Pango

```
1  #include <gtk/gtk.h>
2  #include <iostream>
3  using namespace std;
4  GtkWidget * MainWindow;
5  GtkWidget * DrawingArea;
6  PangoLayout * layout;
7  PangoContext * context;
8  PangoMatrix matrix = PANGO_MATRIX_INIT;
9  PangoFontDescription * font;
10 PangoColor color;
11 double RotationAngle = 0;
12 gchar * FontFamily = NULL;
13 int FontSize = 0;
14 void RedrawSample()
15 {
16       GdkRectangle rect;
```

```
17       rect. x = rect.y = 0;
18       rect.width = 600;
19       rect.height = 500;
20       gdk_window_invalidate_rect
               (DrawingArea->window, &rect, FALSE);
21 }
22 void SetFont(GtkWidget * widget)
23 {
24       GtkWidget * dlg =
                     gtk_font_selection_dialog_new
                                    ("Select font");
25       if (FontFamily)
26          gtk_font_selection_dialog_set_font_name
                    (GTK_FONT_SELECTION_DIALOG(dlg),
                     FontFamily);
27       if (gtk_dialog_run(GTK_DIALOG(dlg))
              == GTK_RESPONSE_OK)
28       {
29             FontFamily =
              gtk_font_selection_dialog_get_font_name
                    (GTK_FONT_SELECTION_DIALOG(dlg));
30             RedrawSample();
31       }
32       gtk_widget_destroy(dlg);
33 }
34 void IncreaseFontSize(GtkWidget * widget)
35 {
36       if (!FontSize)
37          FontSize =
               pango_font_description_get_size(font);
38       FontSize += 1000;
39       RedrawSample();
40 }
41 void DecreaseFontSize(GtkWidget * widget)
42 {
43       if (!FontSize)
44          FontSize =
               pango_font_description_get_size(font);
45       FontSize -= 1000;
46       RedrawSample();
47 }
48 void RotateRight(GtkWidget * widget)
49 {
50       RotationAngle = -10;
51       RedrawSample();
```

```
52 }
53 void RotateLeft(GtkWidget * widget)
54 {
55       RotationAngle = 10;
56       RedrawSample();
57 }
58 void SetColor(GtkWidget * widget)
59 {
60       GdkColor gdk_color;
61       gtk_color_button_get_color
           (GTK_COLOR_BUTTON(widget), &gdk_color);
62       color.red = gdk_color.red;
63       color.green = gdk_color.green;
64       color.blue = gdk_color.blue;
65       RedrawSample();
66 }
67 void Redraw(GtkWidget * widget,
             GdkEventExpose* event, gpointer data)
68 {
69       cout << "Redrawing" << endl;
70       if (FontFamily)
71            cout << "Font family: "
                 << FontFamily << endl;
72       cout << "Font size: "
             << FontSize / 1000 << endl;
73       cout << "Rotation angle: "
             << RotationAngle << endl;
74       context = gtk_widget_get_pango_context
                                    (MainWindow);
75       pango_matrix_rotate(&matrix,
                                  RotationAngle);
76.      pango_context_set_matrix(context,
                                       &matrix);
77       if (FontFamily)
78           font =
               pango_font_description_from_string
                                    (FontFamily);
79       else
80            font =
              pango_font_description_from_string
                                     ("Sans 10");
81       if (FontSize)
82         pango_font_description_set_absolute_size
                                 (font, FontSize);
```

```
83       pango_context_set_font_description
                                    (context, font);
84       PangoAttribute * attr =
              pango_attr_foreground_new
              (color.red, color.green, color.blue);
85       layout = pango_layout_new(context);
86       pango_layout_set_font_description
                                     (layout, font);
87       pango_layout_set_text(layout,
                                  "Sample text", -1);
88       PangoAttrList * list =
                                pango_attr_list_new();
89       pango_attr_list_insert(list, attr);
90       pango_layout_set_attributes(layout, list);
91       gtk_paint_layout(
92          widget->style,
93          widget->window,
94          GTK_STATE_NORMAL,
95          FALSE,
96          &event->area,
97          widget,
98          "drawingarea",
99          250, 250,
100         layout);
101      RotationAngle = 0;
102 }
103 int main(int argc, char ** argv)
104 {
105      gtk_init(&argc, &argv);
106      MainWindow =
                gtk_window_new(GTK_WINDOW_TOPLEVEL);
107      GtkWidget * hBox = gtk_hbox_new(FALSE, 2);
108      GtkWidget * vBox = gtk_vbox_new(FALSE, 2);
109      DrawingArea = gtk_drawing_area_new();
110      GtkWidget *btnFontFamily, *btnFontSizeInc,
                   *btnFontSizeDec,
111.                * btnRotateLeft,
                   *btnRotateRight,
                   *btnColor;
112      btnFontFamily =
                  gtk_button_new_with_label("Font");
113      btnFontSizeInc =
                     gtk_button_new_with_label("+");
114      btnFontSizeDec =
                     gtk_button_new_with_label("-");
```

```
115       btnRotateLeft =
            gtk_button_new_with_label("Rotate Left");
116       btnRotateRight =
           gtk_button_new_with_label("Rotate Right");
117       btnColor = gtk_color_button_new();
118       gtk_box_pack_start(GTK_BOX(hBox),
                      btnFontFamily, TRUE, TRUE, 0);
119       gtk_box_pack_start(GTK_BOX(hBox),
                     btnFontSizeInc, TRUE, TRUE, 0);
120       gtk_box_pack_start(GTK_BOX(hBox),
                     btnFontSizeDec, TRUE, TRUE, 0);
121       gtk_box_pack_start(GTK_BOX(hBox),
                      btnRotateLeft, TRUE, TRUE, 0);
122       gtk_box_pack_start(GTK_BOX(hBox),
                     btnRotateRight, TRUE, TRUE, 0);
123       gtk_box_pack_start(GTK_BOX(hBox),
                           btnColor, TRUE, TRUE, 0);
124       gtk_box_pack_start(GTK_BOX(vBox), hBox,
                                       TRUE, TRUE, 0);
125       gtk_box_pack_start(GTK_BOX(vBox),
                         DrawingArea, TRUE, TRUE, 0);
126       gtk_drawing_area_size
                     (GTK_DRAWING_AREA(DrawingArea),
                      600, 500);
127       gtk_window_set_title
                           (GTK_WINDOW(MainWindow),
                           "Pango demonstration!");
128       gtk_window_resize(GTK_WINDOW(MainWindow),
                                            600, 600);
129       gtk_container_add
                    (GTK_CONTAINER(MainWindow), vBox);
130       g_signal_connect(DrawingArea,
           "expose-event", G_CALLBACK(Redraw), NULL);
131       g_signal_connect(btnFontFamily,
                "clicked", G_CALLBACK(SetFont), NULL);
132       g_signal_connect(btnFontSizeInc,
                "clicked",
                 G_CALLBACK(IncreaseFontSize), NULL);
133       g_signal_connect(btnFontSizeDec,
                "clicked",
                 G_CALLBACK(DecreaseFontSize), NULL);
134       g_signal_connect(btnRotateRight,
                "clicked",
                 G_CALLBACK(RotateRight), NULL);
```

```
135       g_signal_connect(btnRotateLeft,
               "clicked",
                G_CALLBACK(RotateLeft), NULL);
136       g_signal_connect(btnColor, "color-set",
                G_CALLBACK(SetColor), NULL);
137       gtk_widget_show_all(MainWindow);
138       gtk_main();
139       return 0;
140 }
```

В примере использованы следующие типы Pango: `PangoContext`, `PangoLayout`, `PangoFontDescription`, `PangoColor`, `PangoMatrix`. `PangoLayout` — это тип, который представляет параграф текста со всеми атрибутами (шрифт, размер, цвет). Те атрибуты, которые относятся к шрифту, задаются при помощи типа `PangoFontDescription`. Цвет же текста задается как атрибут (тип `PangoAttribute`), который инициализируется при помощи объекта типа `PangoColor`. Тип `PangoMatrix` используется для определения масштаба и угла вращения текста.

Внешний вид программы показан на рисунке 3.2.

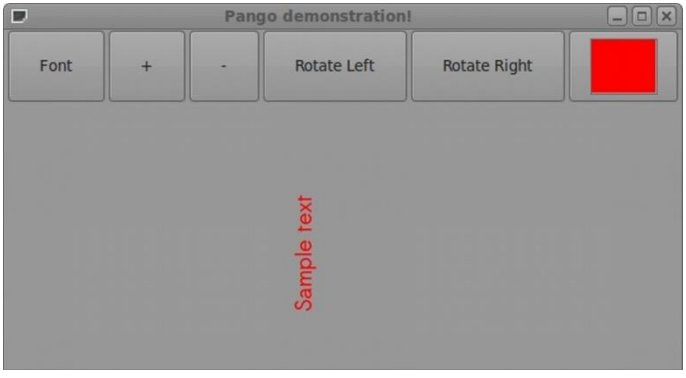

Рис. 3.2. Окно программы gtk_pango

3.2.5. Автоматизация пользовательского интерфейса: GtkUIManager, GtkBuilder, Glade

Как говорилось выше, пользовательский интерфейс представляет собой множество элементов, которые вложены друг в друга, образуя сложную структуру. Разумеется, что для сравнительно больших приложений нецелесообразно создавать и размещать все элементы вручную. Именно для облегчения определения пользовательского интерфейса и созданы следующие возможности. Начнем с самого простого — `GtkUIManager`.

GtkUIManager

`GtkUIManager` — это тип, который создан для облегчения процесса создания меню и панелей инструментов (toolbar). При использовании `GtkUIManager` эти части пользовательского интерфейса задаются в XML-файле. Этот файл обрабатывается во время выполнения программы и в соответствии с данными в файле создается меню и панель инструментов. В листинге 3.8 показан пример XML-документа, который задает меню и панель инструментов.

Листинг 3.8: Создание меню и панели инструментов

```
<ui>
  <menubar name="MenuBar">
      <menu name="FileMenu"
              action="FileMenuAction">
       <menuitem name="New" action="New"/>
       <menuitem name="Open" action="Open"/>
       <menuitem name="Save" action="Save"/>
      </menu>
      <menu name="PrintMenu"
                  action="PrintMenuAction">
       <menuitem name="Print" action="Print"/>
      </menu>
  </menubar>
  <toolbar name="Toolbar">
      <toolitem name="New" action="New"/>
      <toolitem name="Open" action="Open"/>
      <toolitem name="Save" action="Save"/>
      <separator/>
      <toolitem name="Print" action="Print"/>
  </toolbar>
</ui>
```

Тегам <menubar>, <menu>, <menuitem>, <toolbar>, <toolitem> соответствуют типы GtkMenuBar, GtkMenu, GtkMenuItem, GtkToolbar и GtkToolItem и т.д. В листинге 3.9 показано, как можно при помощи такого XML-файла и GtkUIManager автоматизировать создание графического интерфейса для приложения.

Листинг 3.9: Автоматизация интерфейса: UI Manager

```
1  #include <gtk/gtk.h>
2
3  static void New (GtkMenuItem *menuitem,
                    gpointer data) {};
4  static void Open (GtkMenuItem *menuitem,
                     gpointer data) {};
5  static void Save (GtkMenuItem *menuitem,
                     gpointer data) {};
6 static void Print (GtkMenuItem *menuitem,
                     gpointer data) {};
7
8  #define NUM_ENTRIES 6
9  static GtkActionEntry entries[] =
10 {
11   {"FileMenuAction", NULL, "_File",
                        NULL, NULL, NULL},
12   { "New", GTK_STOCK_NEW, NULL, NULL,
13      "Create a new file", G_CALLBACK (New) },
14   { "Open", GTK_STOCK_OPEN, NULL, NULL,
15      "Open an existing file", G_CALLBACK (Open)},
16   { "Save", GTK_STOCK_SAVE, NULL, NULL,
17      "Save the document to a file",
        G_CALLBACK (Save) },
18   { "PrintMenuAction", NULL, "_Print",
                          NULL, NULL, NULL },
19   { "Print", GTK_STOCK_PRINT, NULL, NULL,
20     "Print the document", G_CALLBACK (Print) }
21 };
22
23 int main(int argc, char ** argv)
24 {
25   GtkWidget *MainWindow, *menubar, *toolbar,
               *vbox;
26   GtkActionGroup *group;
```

```
27   GtkUIManager *uimanager;
28
29   gtk_init (&argc, &argv);
30
31   MainWindow = gtk_window_new
                  (GTK_WINDOW_TOPLEVEL);
32   gtk_window_set_title
          (GTK_WINDOW (MainWindow), "UI Manager");
33   gtk_widget_set_size_request(MainWindow,250,-1);
34
35   group = gtk_action_group_new
                 ("MainActionGroup");
36   gtk_action_group_add_actions (group, entries,
                 NUM_ENTRIES, NULL);
37
38   uimanager = gtk_ui_manager_new ();
39   gtk_ui_manager_insert_action_group
                   (uimanager, group, 0);
40   gtk_ui_manager_add_ui_from_file
                   (uimanager, "ui.xml", NULL);
41
42   menubar = gtk_ui_manager_get_widget
                   (uimanager, "/MenuBar");
43   toolbar = gtk_ui_manager_get_widget
                   (uimanager, "/Toolbar");
44   gtk_toolbar_set_style (GTK_TOOLBAR (toolbar),
                          GTK_TOOLBAR_ICONS);
45   gtk_window_add_accel_group
                   (GTK_WINDOW (MainWindow),
                    gtk_ui_manager_get_accel_group
                    (uimanager));
46
47   vbox = gtk_vbox_new (FALSE, 0);
48   gtk_box_pack_start_defaults
                   (GTK_BOX (vbox), menubar);
49   gtk_box_pack_start_defaults
                   (GTK_BOX (vbox), toolbar);
50
51   gtk_container_add
               (GTK_CONTAINER (MainWindow), vbox);
52   gtk_widget_show_all (MainWindow);
53
54   gtk_main ();
55   return 0;
56 }
```

Внешний вид программы представлен на рисунке 3.3.

Рис. 3.3. Окно программы gtk_ui_demo

При использовании `UIManager` для начала нужно определить множество действий, которые должны соответствовать тем действиям, которые были заданы в XML-файле. Для этого в строках 8 - 21 определен массив из элементов типа `GtkActionEntry`, который далее (в строке 36) загружается в объект типа `GtkActionGroup`. Именно этот объект в строке 39 передается объекту `UIManager`. Кроме функциональности `UIManager` в листинге показано использование одного из простых контейнеров — `GtkVBox`. Он позволяет организовать несколько элементов интерфейса один поверх другого.

GtkBuilder

`GtkBuilder` — это тип GTK+, который делает возможным определение всего пользовательского интерфейса через XML-файл. Для начала создается XML-файл, в котором при помощи конкретных тегов, описывается пользовательский интерфейс. Рассмотрим один пример.

Листинг 3.10: Определение пользовательского интерфейса

```
<interface>
  <object class="GtkDialog" id="LoginDialog">
   <property name="title">GTK Builder</property>
    <child internal-child="vbox">
      <object class="GtkVBox" id="vbox1">
        <property name="border-width">10</property>
        <child>
         <object class="GtkTable" id="table1">
               <property name="n-rows">2</property>
               <property name="n-columns">2
                </property>
               <child>
```

```
                    <object class="GtkLabel"
                            id="label1">
                        <property name="label">
                          Login:
                        </property>
                    </object>
                </child>
                <child>
                    <object class="GtkEntry"
                            id="login" />
                    <packing>
                    <property name="left-attach">
                        1
                     </property>
                    </packing>
                </child>
                <child>
                    <object class="GtkLabel"
                            id="label2">
                           <property name="label">
                             Password:
                           </property>
                    </object>
                    <packing>
                    <property name="left-attach">
                        0
                    </property>
                    <property name="top-attach">
                        1
                    </property>
                    </packing>
                </child>
                <child>
                    <object class="GtkEntry"
                            id="password" >
                    <property name="visibility">
                        FALSE
                    </property>
                    </object>
                    <packing>
                    <property name="left-attach">
                        1
                    </property>
```

```
                    <property name="top-attach">
                        1
                    </property>
                    </packing>
                </child>
            </object>
        </child>
        <child internal-child="action_area">
          <object class="GtkHButtonBox"
                  id="hbuttonbox1">
            <property name="border-width">
               20
            </property>
            <child>
              <object class="GtkButton"
                      id="close_button">
                <property name="label">
                      gtk-close
                </property>
                <property name="use-stock">
                      TRUE
                </property>
                <signal name="clicked"
                    handler="close_button_clicked"/>
              </object>
            </child>
            <child>
              <object class="GtkButton"
                      id="ok_button">
                <property name="label">
                      gtk-ok
                </property>
                <property name="use-stock">
                      TRUE
                </property>
                <signal name="clicked"
                       handler="ok_button_clicked"/>
              </object>
            </child>
          </object>
        </child>
      </object>
    </child>
  </object>
</interface>
```

В этом файле описывается интерфейс диалогового окна, который позволяет ввести имя и пароль пользователя. В листинге 3.11 приведен исходный код программы, где показано применение типа `GtkBuilder` при создании интерфейса (в качестве XML-файла использован код из листинга 3.10).

Листинг 3.11: Автоматизация интерфейса: GtkBuilder

```
1  #include <gtk/gtk.h>
2  #include <iostream>
3  using namespace std;
4
5  GtkDialog * MainWindow;
6
7  extern "C" void close_button_clicked
                        (GtkWidget * widget)
8  {
9        cout << "Close clicked" << endl;
10       gtk_main_quit();
11 }
12
13 extern "C" void ok_button_clicked
                        (GtkWidget * widget)
14 {
15       cout << "OK clicked" << endl;
16       gtk_widget_destroy(GTK_WIDGET(MainWindow));
17 }
18
19 int main(int argc, char ** argv)
20 {
21       GtkBuilder * Builder;
22       gtk_init(&argc, &argv);
23       Builder = gtk_builder_new();
24       if (!gtk_builder_add_from_file
              (Builder, "gtk_builder.xml", NULL))
25       {
26             cerr << "Cannot load UI file"
                        << endl;
27             return -1;
28       }
29       gtk_builder_connect_signals
              (Builder, NULL);
```

```
30        MainWindow = GTK_DIALOG
                (gtk_builder_get_object
                    (Builder, "LoginDialog"));
31        gtk_widget_show_all
                (GTK_WIDGET(MainWindow));
32        gtk_main();
33        return 0;
34 }
```

Внешний вид программы показан на рисунке 3.4.

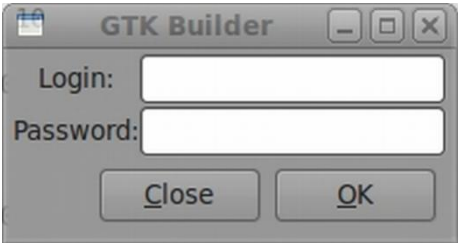

Рис. 3.4. Окно программы gtk_builder

Примечание 3.2.3. При компиляции данного файла необходимо передать компилятору ключ `-export-dynamic`. Без этого ключа обработчики сигналов не будут включены в бинарный файл приложения и `GtkBuilder` не сможет их найти.

Следует отметить, что, поскольку приложение написано на языке C++, а библиотека GTK+ на C, то при определении обработчиков сигналов нужно указать модификатор `extern "C"`. Иначе компилятор модифицирует имена функций и `GtkBuilder` не сможет их найти.

Glade

Разумеется, вручную можно описать интерфейс лишь сравнительно небольших приложений. Но иногда встречаются приложения, которые снабжены десятками XML-файлов, описывающих интерфейс пользователя. В таких случаях используют Glade — инструмент для определения интерфейса. Это приложение, которое предоставляет удобный интерфейс для задания дизайна вашего приложения. При создании проекта Glade нужно в секции "*Project file format*" указать

GtkBuilder. После сохранения создается файл с расширением .glade. Этот файл обычно для удобства сохраняют в каталоге проекта.

3.3. Учебный пример: графический редактор

Для того чтобы закрепить материал по разработке графических приложений GTK, рассмотрим учебный пример: примитивный векторный графический редактор. При разработке этого примера будут использованы GtkBuilder и GtkUIManager — для описания интерфейса, меню, панели инструментов, диалог выбора цвета, средства библиотеки GDK для отрисовки графических объектов, и, наконец, полиморфизм C++ для описания классов графических фигур. Также будут обрабатываться сигналы мыши и будут использованы акселераторы.

Исходный код примера можно найти в каталоге Sources/chapter3/example прилагаемого архива примеров (см. раздел 6).

3.3.1. Постановка задачи

Нужно разработать приложение, которое позволит рисовать элементы векторной графики. Оно должно поддерживать несколько геометрических фигур (отрезок, прямоугольник и эллипс), которые далее могут быть выбраны и удалены или перемещены. Необходимо определить свой собственный формат файла, в котором можно будет сохранить созданные рисунки и из которого можно будет их загружать.

3.3.2. Иерархия графических фигур

Для определения графических фигур используем классы. Будут созданы:

- один абстрактный базовый класс Shape, который содержит основные атрибуты графических фигур (цвет, координаты верхней левой и нижней правой точки, состояние и т.д.),
- несколько функций, которые являются общими и могут быть реализованы в самом базовом классе,
- а также ряд чисто виртуальных методов, которые задают интерфейс всех графических фигур. В этот ряд входят такие функции, как перерисовка, перемещение, изменение размера и т.д.

3.3.3. Сохранение и загрузка рисунков

Для сохранения и загрузки рисунков необходимо определить свой собственный формат файла. В данном примере будет использован текстовый файл следующего формата:

```
line :10,25,50,48:#00ff00 ;
rect :250,32,320,86:#00eeee ;
```

Первые две цифры после имени объекта задают координаты верхней левой точки фигуры, а остальные две — нижней правой точки.

4. СОЗДАНИЕ ГРАФИЧЕСКИХ ПРИЛОЖЕНИЙ: Qt

В данном разделе представлены базовые приёмы разработки приложений на языке C++ с использованием Qt версии 4.

Qt (http://qt.nokia.com/) — набор программных средств разработки приложений для разных платформ: Windows, Linux (в т. ч. и для встраиваемых систем), MacOS X и т. д. В состав Qt входят библиотеки классов, инструменты для разработки пользовательского интерфейса, средства тестирования и многое другое. В настоящее время правами на Qt владеет компания Nokia.

Qt широко используется при разработке как коммерческих, так и открытых программных продуктов. В частности, Qt — основа для приложений из состава desktop-среды KDE (http://kde.org/).

Qt распространяется как под коммерческой лицензией, так и под открытыми: LGPL и GPL.

Данный раздел не претендует на то, чтобы быть справочником по Qt или подробным руководством по разработке Qt-приложений. Здесь показаны базовые техники, необходимые при создании различных программных продуктов с использованием Qt.

4.1. Основные компоненты Qt

В состав Qt входят библиотеки классов, предназначенных для решения самых разных задач, например:

- `QtCore` — базовые средства Qt;
- `QtGui` — классы для создания компонентов графического интерфейса (GUI);
- `QtMultimedia` и `Phonon` — классы для работы с мультимедиа-данными;
- `QtNetwork` — классы для работы с сетью;
- `QtXml` и `QtXmlPatterns` — средства для обработки данных в формате XML;
- `QtTest` — средства для тестирования Qt-приложений;
- и многие другие.

Qt также предоставляет инструменты для проектирования графического интерфейса (`QtDesigner`), для локализации приложений (`QtLinguist`) и др. В состав Qt входит и среда разработки

QtCreator IDE, хотя Qt-приложения можно создавать и без неё. В рассматриваемых ниже примерах не предполагается использование какой-либо специальной среды разработки.

4.2. "Hello World" Qt

В качестве простого примера рассмотрим Qt-приложение, создающее окно со строкой ”Hello World!” в текстовом поле.

Рис. 4.1. Окно приложения «HelloWorldQt»

4.2.1. Структура приложения

Исходный код приложения "HelloWorldQt" представлен в листинге 4.1.

Листинг 4.1: Приложение HelloWorldQt

```
#include <QApplication>
#include <QTextEdit>

int main(int argc, char **argv)
{
    QApplication app(argc, argv);

    QTextEdit textEdit("Hello World!");
    textEdit.show();
    return app.exec();
}
```

"`QApplication`" и "`QTextEdit`" — заголовочные файлы, которые необходимо включить, чтобы можно было использовать соответствующие классы Qt.

Для каждого класса в Qt существует отдельный заголовочный файл с тем же именем. Кроме этого, можно использовать заголовочные файлы для соответствующих компонентов Qt в целом вместо то-

го, чтобы включать в код приложения по файлу для каждого используемого класса. Например:

```
#include <QtGui>
```

В начале работы функция `main()` создаёт объект `QApplication` и передаёт ему параметры командной строки приложения (некоторые из таких параметров могут быть предназначены не самому приложению, а Qt).

Затем создаётся элемент для редактирования текста (объект класса `QTextEdit`), в котором будет отображаться строка "Hello World!". Этот элемент является основным компонентом приложения. Собственно окно приложения, на котором он размещается (со строкой заголовка, необходимыми кнопками и т.д.), будет создано Qt автоматически.

Элементы графического пользовательского интерфейса, располагающиеся в окнах приложения, в Qt принято называть виджетами (widget). Созданный в данном примере элемент для редактирования текста (text edit) — один из примеров виджетов.

Сразу после создания виджет, как правило, не отображается на экране (это верно для виджетов "верхнего уровня", т.е. тех, для которых нет родительских виджетов — подробнее об этом см. ниже). Вызов метода `show()` делает этот виджет видимым.

Чтобы элементы графического интерфейса не просто отображались на экране, но и реагировали на действия пользователя, необходимо запустить цикл обработки событий. Для этого вызывается метод `exec()` объекта-приложения (`app`). Значение, которое возвращает этот метод, обычно используется как возвращаемое значение функции `main()`.

4.2.2. Сборка приложения

Для сборки Qt-приложения, как правило, используется так называемый файл проекта (project file). В нём указывается название приложения, список файлов с исходным кодом, а также, при необходимости, другие данные. Файл проекта, как правило, удобно поместить в каталог, где находятся файлы с исходным кодом.

Для приложения "HelloWorldQt" файл проекта (`hello.pro`) может быть таким:

```
TARGET = HelloWorldQt
SOURCES += hello.cpp
```

В параметре TARGET указано название приложения (такое же имя будет и у исполняемого файла приложения). В параметре SOURCES перечислены файлы с исходным кодом.

С помощью инструмента qmake по этому файлу проекта можно подготовить Makefile для сборки приложения. Для этого достаточно запустить qmake в каталоге, где находится файл проекта, hello.pro.

В полученном Makefile будут все необходимые инструкции для сборки "HelloWorldQt". При сборке будет использоваться компилятор языка C++, для которого библиотеки Qt сконфигурированы на данной системе (обычно — g++). Для того чтобы собрать приложение, теперь достаточно выполнить команду make. В результате будет создан исполняемый файл с именем HelloWorldQt.

Примечание 4.2.1. На некоторых системах инструмент qmake может называться по-другому, например, qmake-qt4.

Примечание 4.2.2. Возможно, удобнее не создавать файл проекта вручную, а генерировать его автоматически, выполнив команду qmake -project в каталоге с исходным кодом приложения.

4.3. Элементы пользовательского интерфейса

Приложения с графическим интерфейсом, использующие Qt, как правило, создают одно или более окон (в т. ч. диалоговых окон), в которых располагаются различные виджеты.

Ниже при описании возможностей Qt в качестве примера рассматривается программа "SimpleAppQt", полный исходный код которой представлен в архиве примеров в каталоге Sources/chapter4/simple_app (см. раздел 6). Приложение "SimpleAppQt" собирается из следующих файлов:

simpleapp.pro	файл проекта для qmake
simpleapp.cpp	реализация функции main()
mainwidget.*	реализация основного виджета приложения
additemdialog.*	реализация диалога "Add item..."
simpleapp.qrc	список ресурсов приложения
images/app.png	значок-иконка для приложения (можно использовать произвольное изображение в формате PNG, имеющее размер не менее 16x16 пикселей)

В основном окне "SimpleAppQt" (см. рис. 4.2) находится "list widget" со списком строковых значений, а также кнопки "*Add...*", "*Clear*" и "*Sort*". Элементу "list widget" соответствует класс `QListWidget`, кнопкам — `QPushButton`.

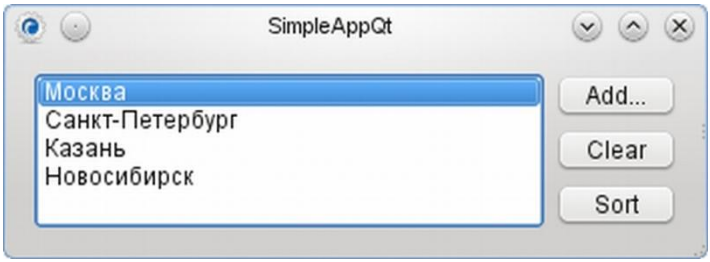

Рис. 4.2. Список элементов в окне приложения “SimpleAppQt”

При нажатии на "*Clear*" список очищается, на "*Sort*" - сортируется по возрастанию. При нажатии на "*Add...*" открывается диалоговое окно "*Add item...*", в котором можно ввести произвольное строковое значение (см. рис. 4.3). Это значение будет добавлено в список в основном окне.

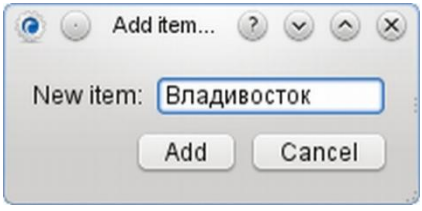

Рис. 4.3. Диалог «Add item» приложения “SimpleAppQt”

Для отображения текста "*New item*" используется объект класса `QLabel`, для создания поля для ввода текста — `QLineEdit`.

Qt (модуль `QtGui`) предоставляет набор стандартных виджетов для самых разных ситуаций. Список таких виджетов можно найти в Qt Reference Documentation [7], а также в инструменте `QtDesigner`.

В данном примере графический интерфейс приложения создаётся напрямую в коде соответствующих методов классов `MainWidget` и `AddItemDialog`. На практике бывает удобнее использовать средства

визуального проектирования графического интерфейса, предоставляемые инструментом `QtDesigner`. Подготовленный в `QtDesigner` интерфейс сохраняется в виде файла в специальном формате, по которому при сборке инструмент `uic` ("user interface compiler") автоматически генерирует файлы с соответствующим кодом на языке C++. Эти файлы впоследствии используются при сборке приложения.

4.4. Система объектов Qt

4.4.1. Общие сведения

Для того чтобы использовать средства Qt в своих приложениях, полезно знать об особенностях системы объектов Qt (Qt Object Model) и о возможностях, которые она предоставляет, в дополнение к реализации ООП в C++, в частности:

- средства для работы с мета-информацией (introspection): для объектов произвольного класса — наследника `QObject` — с их помощью можно получить имя класса, информацию о классах, от которых данный класс наследуется, и т.п.;
- поддержка свойств (`properties`) для классов;
- механизм взаимодействия объектов с помощью т.н. «сигналов» и «слотов» (см. ниже);
- средства для связи объектов по принципу «родитель — потомок» и соответствующего механизма для работы с памятью (например, когда удаляется объект-«родитель», удаляются и все его объекты-«потомки»);
- поддержка локализации строковых значений (т.е. использование перевода строки на требуемый язык вместо значения этой строки по умолчанию).

При сборке приложения с использованием `qmake` и `make`, как описано выше, автоматически запускается инструмент `moc` (meta-object compiler). Он анализирует исходный код приложения, извлекает оттуда необходимую информацию о классах и создаёт C++ файлы, где эта информация представлена в нужном формате. Эти файлы затем компилируются и компонуются так же, как и остальные части приложения.

Для большей части классов Qt (в т. ч. - для всех виджетов) на вершине иерархии классов находится `QObject`. `QObject` использует

мета-информацию, полученную с помощью moc, для реализации возможностей, описанных выше.

В Qt базовым классом для виджетов (как стандартных, так и создаваемых пользователем) является QWidget.

Для примера рассмотрим класс MainWidget из "SimpleAppQt", который отвечает за основное окно приложения. Этот класс является наследником QWidget, а значит, в конечном счёте, и QObject:

```
class MainWidget : public QWidget
{
    Q_OBJECT
    public:
    MainWidget(QWidget *parent = NULL);
    ...
};
```

Q_OBJECT в самом начале объявления класса указывает, что данный класс при сборке приложения должен быть обработан инструментом moc. Этот маркер нужно указывать для всех классов-наследников QObject, которые используют «сигналы» и «слоты», «свойства» (properties) и т.д.

Примечание 4.4.1. Описания классов обычно находятся в заголовочных файлах приложения. Для того чтобы инструмент moc обработал их при сборке этого приложения, они должны быть перечислены в параметре HEADERS файла проекта. При создании файла проекта с помощью команды “qmake -project” это делается автоматически.

Одна из функций системы объектов Qt — поддержка локализации приложений. Как можно заметить в коде "SimpleAppQt", строковые значения там не используются напрямую, а передаются в специальную функцию tr(). Это метод класса QObject, отвечающий за локализацию строк (перевод на нужный язык). tr() для данной строки возвращает её перевод, если он есть, или саму строку, если перевод не найден. Эту функцию стоит использовать для всех непосредственно заданных строковых значений.

Инструменты из состава Qt позволяют автоматически извлечь из исходного кода приложения все строки, для которых вызывается tr(), и создать заготовку для файла с переводом. Собственно перевод нередко удобно выполнять в среде QtLinguist. Используя класс

QTranslator, в программе можно загрузить и использовать подготовленный файл с переводом. Подробная информация о локализации Qt-приложений представлена в Qt Reference Documentation [7].

4.4.2. Отношение «родитель — потомок»

В Qt каждый конструктор класса, как правило, имеет параметр parent. Если при вызове конструктора для объекта этот параметр равен NULL, создаётся объект, не имеющий «родителя» (объект верхнего уровня, top-level object). Если parent не равен NULL, создаваемый объект будет «потомком» (дочерним объектом) для объекта, на который указывает parent.

Стоит различать отношение наследования классов (base — derived) и отношение «родитель — потомок» (parent — child) для объектов. При наследовании классов, объект производного класса *является* в некотором смысле и объектом базового класса. В случае отношения «родитель — потомок» родительский объект *владеет* дочерним объектом. Можно также сказать, что «родитель» *отвечает за* своих «потомков». В частности, если для виджета-«родителя» вызывается метод show(), чтобы отобразить его на экране, show() будет вызван автоматически и для дочерних виджетов. Аналогично - при вызове метода hide() для того, чтобы убрать виджет с экрана.

Основная особенность отношения «родитель — потомок» связана с удалением объектов. Когда удаляется родительский объект (т.е. когда выполняется соответствующий деструктор), автоматически удаляются и все его дочерние объекты (для указателей на них вызывается delete). Это даёт возможность не вызывать delete для этих объектов явно.

Например, конструктор класса MainWidget из "SimpleAppQt" выглядит так:

```
MainWidget::MainWidget(QWidget *parent)
: QWidget(parent)
{
    createUI();
    makeConnections();
}
```

В createUI() создаются все виджеты для основного окна приложения, которое для них и устанавливается в качестве «родителя»

(так как значение соответствующего параметра конструкторов `QPushButton`, `QListWidget` и т.д. равно `this`), например:

```
buttonAdd = new QPushButton(tr("Add..."), this);
buttonClear = new QPushButton(tr("Clear"), this);
...
listWidget = new QListWidget(this);
```

Определять деструктор для `MainWidget` в данном случае нет необходимости, так как все ресурсы, выделяемые объектом класса `MainWidget`, — это только дочерние объекты, а они будут удалены автоматически.

Стоит заметить, что если уничтожается объект, у которого есть «родитель», то данный объект автоматически удаляется из списка «потомков» этого «родителя». То есть допустимо удалять дочерние объекты и явно: при удалении «родителя» `delete` для них вызван уже не будет.

В Qt-приложениях, как правило, используется два варианта создания объектов.

1. Объект создаётся *динамически* (т.е. с помощью `new`) и для него указывается родительский объект. Этот вариант используется в приведённом выше примере. При удалении родительского объекта будет удалён и этот дочерний объект.
2. Объект создаётся *на стеке* (т.е. локально для данной области видимости, scope). По правилам C++, при выходе из области видимости такой объект удаляется автоматически. Родительский объект в таких случаях указывается не всегда.

На стеке обычно создаются объекты верхнего уровня, например, основное окно приложения (см. `main()` для "SimpleAppQt" и "HelloWorldQt"). Родительского объекта такие объекты не имеют.

Модальные диалоговые окна также нередко создаются на стеке (и родительский объект для них указывается). Ниже приведён фрагмент кода "SimpleAppQt", в котором создаётся модальный диалог "*Add item...*":

```
void MainWidget::addItem() {
   AddItemDialog addItemDialog(this);
   if (addItemDialog.exec() == QDialog::Accepted) {
      listWidget->
         addItem(addItemDialog.getItemText());
   }
}
```

Когда пользователь закроет диалог "*Add item...*", нажав на кнопку "*Add*", или "*Cancel*", или кнопку закрытия в заголовке окна, `addItemDialog.exec()` вернёт управление. Если пользователь нажал "*Add*" (при этом результат `exec()` будет равен `QDialog::Accepted`), содержимое текстового поля диалога будет добавлено в список строк в основном окне приложения. При выходе из `MainWidget::addItem()` объект `addItemDialog` будет уничтожен.

4.5. Взаимодействие объектов: сигналы и слоты

При разработке приложений бывает необходимо сделать так, чтобы одни объекты реагировали на события, происходящие с другими объектами. Например, в случае "SimpleAppQt" нужно, чтобы при нажатии на кнопку "*Clear*" был очищен список строк, отображаемый приложением, чтобы кнопка "*Add*" в диалоге "*Add item...*" была доступна только тогда, когда в текстовом поле есть текст и т.д.

Qt предоставляет механизм для организации такого взаимодействия объектов классов, являющихся производными от `QObject`: механизм *сигналов* и *слотов* ("signals and slots").

Примечание 4.5.1. Помимо этого механизма в Qt также есть средства и для работы с событиями такими, как нажатие пользователем кнопок мыши и т.д. Подробную информацию об обработке событий в Qt-приложениях можно найти в Qt Reference Documentation [7]. Ниже рассматривается только взаимодействие объектов посредством сигналов и слотов.

Примечание 4.5.2. Сигналы, используемые в Qt, и сигналы, как они обычно понимаются в Unix-системах (`SIGSEGV`, `SIGTERM` и т.д.), — это понятия, не связанные между собой. В данном разделе рассматриваются только сигналы Qt.

Механизм сигналов и слотов работает следующим образом. Сигнал, посылаемый одним объектом, можно связать со слотом (функцией) в другом объекте. Эта функция будет вызываться каждый раз, когда будет послан этот сигнал. Связь сигнала со слотом выполняется с помощью метода `connect()` класса `QObject`:

```
connect(объект-источник, SIGNAL(сигнал()),
        объект-приёмник, SLOT(слот()));
```

В документации по Qt для каждого класса приводится описание сигналов, которые посылаются объектами этого класса, а также описание реализованных в классе слотов.

Например, в "SimpleAppQt" в функции `MainWidget::makeConnections()` сигнал `clicked()`, посылаемый объектом `buttonAdd` (класс `QPushButton`), подключается к слоту `addItem()` объекта класса `MainWidget`. Таким образом, каждый раз, когда пользователь нажимает кнопку "*Add...*" в основном окне приложения, в результате выполняется метод `addItem()` для объекта, отвечающего за это окно.

```
connect(buttonAdd, SIGNAL(clicked()),
        this, SLOT(addItem()));
```

Аналогично, с помощью такого механизма взаимодействия несложно добиться того, чтобы при нажатии на кнопку "*Clear*" очищался список строк в элементе "list widget" (`clear()` — стандартный слот, определённый в классе `QListWidget`):

```
connect(buttonClear, SIGNAL(clicked()),
        listWidget, SLOT(clear()));
```

У сигналов и слотов могут быть параметры, как у обычных функций. Например, в реализации диалога "*Add item...*" используется сигнал `textChanged(const QString&)`, посылаемый элементом для редактирования текста, когда этот текст меняется. Новый текст передаётся в качестве параметра сигнала.

```
connect(lineEdit,
        SIGNAL(textChanged(const QString &)),
        this,
        SLOT(enableAddButton(const QString &)));
```

Сигнал подключается к слоту `enableAddButton(const QString &)`, реализованному в этом диалоге. `enableAddButton()`

делает кнопку "*Add*" доступной или недоступной (disabled, grayed) в зависимости от того, пусто ли поле для редактирования текста:

```
void AddItemDialog::enableAddButton (
     const QString &text)
{
    buttonAdd->setEnabled(!text.isEmpty());
}
```

Примечание 4.5.3. При подключении сигналов к слотам указываются только типы параметров, но не имена и не значения.

Примечание 4.5.4. Если у сигнала `n` параметров, а у слота — `m` (`m < n`), причём типы первых `m` параметров сигнала соответствуют типам параметров слота, то сигнал тоже можно подключить к этому слоту. «Лишние» параметры сигнала слотом просто не используются. Например, сигнал `foo(double, int)` можно подключить к слоту `bar(double)` и к слоту `baz()`, не имеющему параметров вообще. Обратное неверно: нельзя подключить сигнал к слоту с б*о*льшим числом параметров, например, `clicked()` к `bar(double)`.

Слот — специальный метод класса, производного от `QObject`. В объявлении класса слоты отмечаются ключевым словом `slots` после `public`, `protected` или `private`, например:

```
class AddItemDialog : public QDialog
{
    Q_OBJECT
    private slots:
        void enableAddButton(const QString &text);
...
};
```

За исключением того, что слот может быть подключен к сигналу, это такой же метод класса, как и остальные. В частности:

- при наследовании классов к слотам применяются те же правила, что и к другим методам;
- слот можно вызывать напрямую, как обычный метод класса;
- слот может быть виртуальным;
- слот может возвращать значение. Оно игнорируется, если слот вызван при получении сигнала, но может быть использовано, если слот вызывается напрямую.

В отличие от слотов сигналы в классе только объявляются (в секции `signals`), но не определяются. Определение будет создано автоматически при обработке исходного кода приложения инструментом `moc`. Тип возвращаемого значения для сигналов — всегда `void`. Пример:

```
class MyGreatNotifier : public QWidget
{
   Q_OBJECT
   ...
   signals:
       void somethingReallyGreatHappened
            (const MyData &);
   ...
};
```

Сигнал посылается с помощью специальной операции `emit`, например:

```
void MyGreatNotifier::foo(const MyData &data)
{
   // Notify all listeners
   emit somethingReallyGreatHappened(data);
   ...
}
```

Сигнал может быть подключен к нескольким слотам одновременно, в том числе и к нескольким слотам одного и того же объекта. В этом случае при сигнале будут последовательно выполнены все соответствующие слоты (Qt не определяет, в каком именно порядке).

Несколько сигналов могут быть подключены к одному и тому же слоту. При этом слот будет вызываться, если послан какой-либо из этих сигналов.

Взаимодействие "сигнал—слот" обычно синхронное, хотя бывают и исключения, в т. ч. в многопоточных приложениях. В приведённом выше примере функция `MyGreatNotifier::foo()` посылает сигнал `somethingReallyGreatHappened()`. В результате последовательно выполняются все слоты, подключенные к этому сигналу, и только после этого функция `MyGreatNotifier::foo()` продолжает работу.

Qt предоставляет средство и для отключения сигналов от слотов (`QObject::disconnect`). На практике оно используются в редких случаях: связь "сигнал—слот" для объектов и так автоматически разрывается, когда уничтожается любой из этих объектов.

4.6. Размеры и расположение элементов интерфейса

Вместо того, чтобы задавать размер и положение каждого виджета явно, в Qt-приложениях применяются так называемые "layout"-объекты. В примере "SimpleAppQt" используются объекты классов `QHBoxLayout` и `QVBoxLayout` для горизонтального и вертикального расположения элементов интерфейса, соответственно. "layout"-объект управляет и размерами находящихся в нём виджетов, в т. ч. отвечает за корректное изменение этих размеров при изменении размеров окна приложения.

В методе `createUI()` класса `MainWidget` кнопки для основного окна "SimpleAppQt" помещаются в `QVBoxLayout`:

```
buttonLayout = new QVBoxLayout();
buttonLayout->addWidget(buttonAdd);
buttonLayout->addWidget(buttonClear);
buttonLayout->addWidget(buttonSort);
```

Виджет для вывода списка строк (объект класса `QListWidget`) и этот "layout"-объект затем располагаются горизонтально с помощью ещё одного "layout"-объекта, который устанавливается в качестве основного для окна приложения с помощью `setLayout()`:

```
mainLayout = new QHBoxLayout();
mainLayout->addWidget(listWidget);
mainLayout->addLayout(buttonLayout);
setLayout(mainLayout);
```

Примечание 4.6.1. При помещении объекта (виджета или другого "layout"-объекта) в "layout"-объект последний автоматически становится «родителем» для первого. При установке "layout"-объекта в качестве основного для виджета, этот виджет становится «родителем» данного "layout"-объекта. Т.е. в данном случае при удалении основного окна "SimpleAppQt" будут автоматически удалены и соответствующие "layout"-объекты, и все виджеты, которые в них были добавлены.

4.7. Особенности классов-контейнеров в Qt

Во многих Qt-приложениях, как и в "SimpleAppQt", для хранения и обработки строковых значений используются объекты класса `QString`. Это один из примеров классов-контейнеров.

Qt предоставляет классы-контейнеры для работы со строками (`QString`), с массивами и списками (`QList<T>`, `QLinkedList<T>` и др.), с множествами и ассоциативными массивами (`QSet<T>`, `QMap<T, U>` и др.) и многие другие. Подробное описание классов-контейнеров, итераторов и соответствующих алгоритмов представлено в Qt Reference Documentation. В данном разделе стоит отметить некоторые особенности Qt-контейнеров.

- Классы-контейнеры в Qt не являются производными от класса `QObject` и, соответственно, не участвуют в отношениях «родитель — потомок», не посылают сигналов и не реализуют слоты и т.д.
- В Qt-приложениях можно использовать и классы-контейнеры, предоставляемые другими библиотеками, например, STL. Для удобства в Qt есть средства для преобразования контейнеров STL в Qt-контейнеры и наоборот. Как правило, это методы вида `fromStd*` и `toStd*`, например: `QList::toStdList()`, `QString::toStdString()` и др.
- При копировании Qt-контейнера используется подход "implicit sharing": данные, хранящиеся в контейнере, копируются только тогда, когда какой-то из объектов их меняет ("copy on write"). Это может помочь сэкономить память и избежать потерь производительности приложения. Пример:

```
QString hello("Hello World");
QString hail;
// оба объекта пока ссылаются на одну и ту же строку
hail = hello;
// копирования не было, доп. память не выделялась
hail += " of Qt";
// теперь строка "Hello World" скопирована в объект
hail и к ней добавлено " of Qt". Объект hello при этом
не изменился.
```

- `QString` использует многобайтную кодировку символов в качестве внутреннего представления строки. Т.е. не стоит полагаться на то, что размер одного символа — один байт. Qt предоставляет средства для перевода строк из одной коди-

ровки в другую, например, `QString::toUtf8`, `QString::toLocal8Bit()`, `QString::fromAscii()` и т.д.

4.8. Работа с ресурсами приложений

Приложения с графическим пользовательским интерфейсом нередко используют различные значки (иконки, icons), изображения-заставки (splash screens) и другие ресурсы. Бывает удобно хранить такие файлы непосредственно в исполняемом файле приложения. Qt предоставляет механизм для работы с подобными ресурсами, общий для всех поддерживаемых платформ.

В примере "SimpleAppQt" таким образом хранится значок (иконка) для приложения. В каталоге, где находится файл проекта и файлы с исходным кодом, также есть файл `simpleapp.qrc`. В нём перечислены необходимые ресурсы — PNG-файл `app.png` из подкаталога `images`:

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
<file>images/app.png</file>
</qresource>
</RCC>
```

В файле проекта `simpleapp.qrc` указан в списке значений параметра `RESOURCES`. При сборке приложения файл `simpleapp.qrc` обрабатывается инструментом `rcc` (resource compiler) и перечисленные в нём ресурсы автоматически помещаются в исполняемый файл приложения.

В самом приложении работа с ресурсами выполняется почти так же, как и с любыми другими файлами, только «путь» к файлу из ресурсов начинается с `":/"`. Например, загрузка и установка основного значка приложения выполняется так:

```
QIcon appIcon(":/images/app.png");
app.setWindowIcon(appIcon);
```

Если бы значок нужно было загрузить не из ресурсов, хранящихся в файле приложения, а из обычного файла (допустим, `"/home/user/default.png"`), в коде достаточно было бы просто задать соответствующий путь:

```
QIcon appIcon("/home/user/default.png");
app.setWindowIcon(appIcon);
```

Как правило, если какой-либо метод класса в Qt принимает путь к файлу как параметр, можно в качестве значения этого параметра передавать и путь к файлу из ресурсов приложения.

4.9. Учебный пример: улучшенный текстовый редактор

В качестве учебной задачи предлагается усовершенствовать текстовый редактор из примера "Text Edit" (он же — Rich Text, http://doc.qt.nokia.com/latest/demos-textedit.html), который поставляется вместе с Qt. Исходный вариант этого приложения даёт возможность работать с файлами в формате HTML, редактировать их текст, использовать выделение различными шрифтами, цветами и т.д.

Предлагается добавить в это приложение поддержку работы с изображениями, а именно:

1. Создать панель инструментов с одной кнопкой: «*Вставить изображение...*».
2. Установить клавиатурное сочетание для вставки изображения, например, Ctrl-Alt-I.
3. Создать пункт «*Вставка*» ("*Insert*") в главном меню приложения и в этом меню - пункт «*Вставить изображение...*».
4. Реализовать обработку запросов пользователя. Если требуется вставить изображение в документ, нужно открыть стандартный диалог выбора файла (формат: PNG, JPEG), загрузить выбранный пользователем файл и вставить изображение в текущую позицию в документе.

При вставке изображения в документ (`QTextDocument`) нужно не просто загрузить это изображение из файла, но и добавить его в документ как `ImageResource`, например:

```
QImage image(f);
if (image.isNull()) {
...
// Сообщить, что изображение загрузить не удалось
}

QUrl imageUrl = QUrl::fromLocalFile(f);
textEdit->document()->
       addResource(QTextDocument::ImageResource,
                   imageUrl, QVariant(image));
```

В этом фрагменте кода предполагается, что путь к файлу с изображением хранится в переменной `f`.

Для выполнения задания, помимо описанных выше приёмов разработки Qt-приложений, потребуется также познакомиться с описанием классов `QTextEdit`, `QTextCursor`, `QTextImageFormat` и др., а также с разделом "*Rich Text Processing*" в Qt Reference Documentation (http://doc.qt.nokia.com/latest/richtext.html).

Меню, панель инструментов и объекты-действия (`QAction`) можно подготовить по аналогии с тем, что уже есть в примере "*Text Edit*".

5. Рекомендуемая литература

1. Таненбаум Э. Современные операционные системы. - 3-е изд. - СПб.: Питер, 2010. - 1120 с. - ISBN 978-5-49807-306-4.
2. Курячий Г. В., Маслинский К. А. Операционная система Linux: Курс лекций: учебное пособие. - 2-е изд., испр. – М.: ALT Linux; Издательство ДМК Пресс, 2010. - 347 с. - ISBN 978-5-94074-591-4
3. Таненбаум Э., Вудхалл А. Операционные системы. Разработка и реализация. - 3-е изд. - СПб.: Питер, 2007. – 704 с. - ISBN 978-5-469-01403-4.
4. Мэтью Н., Стоунс Р.. Основы программирования в Linux. - 4-е изд. – СПб.: БХВ-Петербург, 2009. – 896 с. – ISBN 978-5-9775-0289-4.
5. GTK+ Reference Manual. http://library.gnome.org/devel/gtk/.
6. GTK+ Tutorial. http://library.gnome.org/devel/gtk-tutorial/
7. Qt Reference Documentation. http://doc.qt.nokia.com/latest/index.html.
8. Qt Developer Network. http://developer.qt.nokia.com/
9. Jasmin Blanchette, Mark Summerfield, C++ GUI Programming with Qt 4, 2nd Edition. Prentice Hall, 2008, ISBN 0-13-235416-0.

6. Архив примеров

Архив примеров к настоящему учебному пособию можно скачать по ссылке:

http://linuxtesting.org/downloads/edu/examples2011.tar.gz.

7. Комментарии

Комментарии и замечания по тексту пособия и файлам примеров просьба направлять по адресу rubanov@linuxtesting.org.

Учебное издание

Мартиросян Ваграм Артурович
Рубанов Владимир Васильевич
Шатохин Евгений Александрович

ОСНОВЫ ПРОГРАММИРОВАНИЯ В СРЕДЕ ОС LINUX

Редактор: *Л.В. Себова*

Подписано в печать 16.03.2010. Формат 60 × 84 $^{1}/_{16}$.
Усл. печ. л.7,25. Уч.-изд. л. 7,0. Тираж 200 экз. Заказ № -

Учреждение Российской академии наук
«Институт системного программирования РАН»

Государственное образовательное учреждение высшего профессионального образования «Московский физико-технический институт (государственный университет)»

141700, Московская обл., г. Долгопрудный, Институтский пер., 9
E-mail: rio@mail.mipt.ru

Отпечатано ООО «Азбука–2000»
109544, г. Москва, ул. Рабочая, д. 84

ОСНОВЫ ПРОГРАММИРОВАНИЯ В СРЕДЕ ОС LINUX

Учебное пособие посвящено основам программирования в среде операционной системы Linux. Включает основные теоретические сведения и практические учебные примеры и задания, нацеленные на выработку базовых навыков программирования на уровне системных вызовов Linux, а также навыков разработки простых программ с графическим интерфейсом пользователя с использованием библиотек GTK и Qt.

Предназначено для студентов, аспирантов и научных сотрудников для обучения основным инструментам и навыкам программирования в среде ОС Linux.

ISBN 5-7417-0348-8