# 1 Введение и основополагающие концепции

Эта книга рассказывает о *системном программировании*, то есть написании *системного программного обеспечения*. Системные программы работают на нижнем уровне, взаимодействуя непосредственно с ядром и основными системными библиотеками. Ваши командная оболочка и текстовый редактор, компилятор и отладчик, основные утилиты и системные демоны — все это системное программное обеспечение. К данной категории относятся также сетевой сервер, веб-сервер и база данных. Эти компоненты являются классическими образцами системного ПО и взаимодействуют в основном, а то и исключительно с ядром и библиотекой C. Другое программное обеспечение (например, прикладные программы с графическими пользовательскими интерфейсами) находятся на более высоком уровне и взаимодействуют с низкоуровневыми лишь эпизодически. Некоторые специалисты целыми днями пишут только системное программное обеспечение, другие уделяют таким задачам лишь часть рабочего времени, но понимание системного ПО — это навык, который пригодится любому специалисту. Независимо от того, является такое программирование насущным хлебом конкретного инженера либо просто базисом для создания более высокоуровневых концепций, системное программирование — это сердце всего создаваемого нами софта.

Эта книга посвящена не всему системному программированию, а системному программированию в *Linux*. Linux — это современная UNIX-подобная операционная система, написанная с нуля Линусом Торвальдсом и стихийным сообществом программистов со всего мира. Linux разделяет цели и философию UNIX, но при этом не является лишь разновидностью UNIX, а идет своим путем, возвращаясь в русло UNIX, где это желательно, и отклоняясь, когда сие целесообразно. Однако, если не считать базового строения, Linux довольно самобытна. По сравнению с традиционными системами UNIX Linux поддерживает ряд дополнительных системных вызовов, работает иначе и предлагает новые возможности.

## Системное программирование

Программирование для UNIX изначально зарождалось именно как системное. Исторически системы UNIX не включали значительного количества высокоуров-

невых концепций. Даже при программировании в среде разработки, например в системе XWindow, в полной мере задействовались системные API ядра UNIX. Соответственно, можно сказать, что эта книга — о программировании для Linux вообще. Однако учтите, что в книге не рассматриваются *среды разработки* для Linux, например вообще не затрагивается тема `make`. Основное содержание книги — это API системного программирования, предоставляемые для использования на современной машине Linux.

Можно сравнить системное программирование с программированием приложений — и мы сразу заметим как значительное сходство, так и важные различия этих областей. Важная черта системного программирования заключается в том, что программист, специализирующийся в этой области, должен обладать глубокими знаниями оборудования и операционной системы, с которыми он имеет дело. Системные программы взаимодействуют в первую очередь с ядром и системными библиотеками, а прикладные опираются и на высокоуровневые библиотеки. Такие высокоуровневые библиотеки *абстрагируют* детальные характеристики оборудования и операционной системы. У подобного абстрагирования есть несколько целей: переносимость между различными системами, совместимость с разными версиями этих систем, создание удобного в использовании (либо более мощного, либо и то и другое) высокоуровневого инструментария. Соотношение, насколько активно конкретное приложение использует высокоуровневые библиотеки и насколько — систему, зависит от уровня стека, для которого было написано приложение. Некоторые приложения создаются для взаимодействия исключительно с высокоуровневыми абстракциями. Однако даже такие абстракции, весьма отдаленные от самых низких уровней системы, лучше всего получаются у специалиста, имеющего навыки системного программирования. Те же проверенные методы и понимание базовой системы обеспечивают более информативное и разумное программирование для всех уровней стека.

## Зачем изучать системное программирование

В течение прошедшего десятилетия в написании приложений наблюдалась тенденция к уходу от системного программирования к высокоуровневой разработке. Это делалось как с помощью веб-инструментов (например, JavaScript), так и посредством управляемого кода (Java). Тем не менее такие разработки не свидетельствуют об отмирании системного программирования. Действительно, ведь кому-то приходится писать и интерпретатор JavaScript, и виртуальную машину Java, которые создаются именно на уровне системного программирования. Более того, даже разработчики, которые программируют на Python, Ruby или Scala, только выиграют от знаний в области системного программирования, поскольку будут понимать всю подноготную машины. Качество кода при этом гарантированно улучшится независимо от части стека, для которой он будет создаваться.

Несмотря на описанную тенденцию в программировании приложений, большая часть кода для UNIX и Linux по-прежнему создается на системном уровне. Этот код написан преимущественно на C и C++ и существует в основном на базе интерфейсов,

предоставляемых библиотекой C и ядром. Это традиционное системное программирование с применением Apache, `bash`, `cp`, Emacs, `init`, `gcc`, `gdb`, `glibc`, `ls`, `mv`, `vim` и X. В обозримом будущем эти приложения не сойдут со сцены.

К области системного программирования часто относят и разработку ядра или как минимум написание драйверов устройств. Однако эта книга, как и большинство работ по системному программированию, никак не касается разработки ядра. Вместо этого она рассматривает системное программирование для пользовательского пространства, то есть уровень, который находится выше ядра. Тем не менее знания о ядре будут полезным дополнительным багажом при чтении последующего текста. Написание драйверов устройств — это большая и объемная тема, которая подробно описана в книгах, посвященных конкретно данному вопросу.

Что такое системный интерфейс и как я пишу системные приложения для Linux? Что именно при этом мне предоставляют ядро и библиотека C? Как мне удается создавать оптимальный код, какие приемы возможны в Linux? Какие интересные системные вызовы есть в Linux, но отсутствуют в других UNIX-подобных системах? Как все это работает? Именно эти вопросы составляют суть данной книги.

## Краеугольные камни системного программирования

В системном программировании для Linux можно выделить три основных краеугольных камня — системные вызовы, библиотеку C и компилятор C. О каждом из этих феноменов следует рассказать отдельно.

## Системные вызовы

*Системные вызовы* — это начало и конец системного программирования. Системные вызовы (в англоязычной литературе встречается сокращение *syscall*) — это вызовы функций, совершаемые из пользовательского пространства. Они направлены из приложений (например, текстового редактора или вашей любимой игры) к ядру. Смысл системного вызова — запросить у операционной системы определенную службу или ресурс. Системные вызовы включают как всем знакомые операции, например `read()` и `write()`, так и довольно экзотические, в частности `get_thread_area()` и `set_tid_address()`.

В Linux реализуется гораздо меньше системных вызовов, чем в ядрах большинства других операционных систем. Например, в системах с архитектурой x86-64 таких вызовов насчитывается около 300 — сравните это с Microsoft Windows, где предположительно задействуются тысячи подобных вызовов. При работе с ядром Linux каждая машинная архитектура (например, Alpha, x86-64 или PowerPC) может дополнять этот стандартный набор системных вызовов своими собственными. Следовательно, системные вызовы, доступные в конкретной архитектуре, могут отличаться от доступных в другой. Тем не менее большинство системных вызовов — более 90 % — реализованы на всех архитектурах. К этим разделяемым 90 % относятся и общие интерфейсы, о которых мы поговорим в данной книге.

**Обращение к системным вызовам.** Невозможно напрямую связать приложения пользовательского пространства с пространством ядра. По соображениям безопасности и надежности приложениям пользовательского пространства нельзя разрешать непосредственно исполнять код ядра или манипулировать данными ядра. Вместо этого ядро должно предоставлять механизм, с помощью которого пользовательские приложения будут «сигнализировать» ядру о требовании активизировать системный вызов. После этого приложение сможет осуществить *системное прерывание ядра* («ловушка») в соответствии с этим строго определенным механизмом и выполнить только тот код, который разрешит выполнить ядро. Детали этого механизма в разных архитектурах немного различаются. Например, в процессорах i386 пользовательское приложение выполняет инструкцию программного прерывания `int` со значением `0x80`. Эта инструкция осуществляет переключение на работу с пространством ядра — защищенной областью, — где ядром выполняется обработчик программного прерывания. Что же такое обработчик прерывания `0x80`? Это не что иное, как обработчик системного вызова!

Приложение сообщает ядру, какой системный вызов требуется выполнить и с какими параметрами. Это делается посредством *аппаратных регистров*. Системные вызовы обозначаются по номерам, начиная с `0`. В архитектуре i386, чтобы запросить системный вызов `5` (обычно это вызов `open()`), пользовательское приложение записывает `5` в регистр `eax`, после чего выдает инструкцию `int`.

Схожим образом передаются и параметры. Так, в архитектуре i386 каждому из параметров выделяется по регистру — регистры `ebx`, `ecx`, `edx`, `esi` и `edi` в таком же порядке содержат первые пять параметров. В редких случаях, когда системный вызов имеет более пяти параметров, всего один регистр служит указателем на буфер в пользовательском пространстве, где хранятся все эти параметры. Разумеется, у большинства системных вызовов имеется всего пара параметров.

В других архитектурах активация системных вызовов обрабатывается иначе, хотя принцип остается тем же. Вам как системному программисту обычно не нужно знать, как именно ядро обрабатывает системные вызовы. Эта информация уже интегрирована в стандартные соглашения вызова, соблюдаемые в конкретной архитектуре, и автоматически обрабатывается компилятором и библиотекой C.

## Библиотека C

Библиотека C (`libc`) — это основа всех приложений UNIX. Даже если вы программируете на другом языке, то библиотека C, скорее всего, при этом задействуется. Она обернута более высокоуровневыми библиотеками и обеспечивает доступ к службам ядра и системным вызовам. В современных системах Linux библиотека C предоставляется в форме `GNUlibc`, сокращенно `glibc` (произносится как «джи-либ-си», реже «глиб-си»).

Библиотека GNU C предоставляет гораздо больше возможностей, чем может показаться из ее названия. Кроме реализации стандартной библиотеки C, `glibc` дает обертки для системных вызовов, поддерживает работу с потоками и основные функции приложений.

## Компилятор C

В Linux стандартный компилятор языка C предоставляется в форме коллекции *компиляторов GNU* (GNU Compiler Collection, сокращенно `gcc`). Изначально `gcc` представляла собой версию `cc` (компилятора C) для GNU. Соответственно `gcc` расшифровывалась как GNU C Compiler. Однако впоследствии добавилась поддержка других языков, поэтому сегодня `gcc` служит общим названием всего семейства компиляторов GNU. При этом `gcc` — это еще и двоичный файл, используемый для вызова компилятора C. В этой книге, говоря о `gcc`, я, как правило, имею в виду программу `gcc`, если из контекста не следует иное.

Компилятор, используемый в UNIX-подобных системах, в частности в Linux, имеет огромное значение для системного программирования, поскольку помогает внедрять стандарт языка C (см. подраздел «Стандарты языка C» раздела «Стандарты»), а также системный двоичный интерфейс приложений (см. раздел «API и ABI» текущей главы), о которых будет рассказано далее.

**C++**

В этой главе речь пойдет в основном о языке C — лингва франка системного программирования. Однако C++ также играет важную роль.

В настоящее время место C в системном программировании занял C++. Исторически разработчики Linux всегда отдавали C предпочтение перед C++: основные библиотеки, демоны, утилиты и, разумеется, ядро Linux написаны на C. Поэтому, несмотря на доминирование C++ как «улучшенного C» на большинстве не-Linux платформ, в Linux он занимет подчиненное место после C

Тем не менее далее в тексте в большинстве случаев вы можете заменять «C» на «C++». Действительно, C++ — отличная альтернатива C, подходящая для решения практически любых задач в области системного программирования. C++ может связываться с кодом на C, обращаться к системным вызовам Linux, использовать `glibc`.

При использовании C++ в основу системного программирования закладывается еще два краеугольных камня — стандартная библиотека C++ и компилятор GNUC++. *Стандартная библиотека C++* реализует системные интерфейсы C++ и использует стандарт ISOC++ 11. Он обеспечивается библиотекой `libstdc++` (иногда используется название `libstdcxx`). *Компилятор GNUC++* — это стандартный компилятор для кода на языке C++ в системах Linux. Он предоставляется в двоичном файле `g++`.

## API и ABI

Разумеется, программист заинтересован, чтобы его код работал во всех системах, которые планируется поддерживать, как в настоящем, так и в будущем. Хочется быть уверенными, что программы, создаваемые в определенном дистрибутиве Linux, будут работать в других дистрибутивах, а также иных поддерживаемых архитектурах Linux и более новых (а также ранних) версиях Linux.