Содержание

Сокращения и другие соглашения

В таблице ниже перечислены постоянно используемые как в данном пособии, так и в отрасли сокращения и другие речевые соглашения.

сокращение	значение
ПО	Программное обеспечение
ОС	Операционная система

> *Вертикальной чертой слева и курсивом помечены исторические подробности, не требующиеся для понимания курса.*

Тонкой точечной окантовкой и курсивом помечены цитаты.

Названия команд и, по возможности, их параметры для стандартной текстовой оболочки Windows (CMD.EXE) даются ВЕРХНИМ РЕГИСТРОМ, а для текстовых оболочек *nix — нижним регистром (точнее, в единственно возможном для исполнения команд виде, так как в *nix регистр символов учитывается). Примеры команд шеллов могут даваться, при необходимости, со префиксами следующих видов:

`C:\>`	для команд CMD.EXE
`$`	для команд *nix, доступных обычным пользователям
`#`	для команд *nix, предназначенных для выполнения от имени суперпользователя (root)

Вводная. История развития и макрокомпоненты ОС. Монолитное ядро, микроядро, гибридное ядро

Интерфейс взаимодействия с компьютером, по определению, появился вместе с самим компьютером. Изначально он был механическим: оператор вручную выставлял органы управления в нужное состояние (например, крутил колёса, задавая входные числа и программы), а затем запускал составленную программу на выполнение. Позднее появились более совершенные методы ввода-вывода: перфокарты и перфоленты, магнитные ленты, магнитные диски… Но суть оставалась прежней: сначала математически формулировалась задача, затем составлялся алгоритм её решения, потом этот алгоритм переводился в программу, которую уже оператор вводил в компьютер, следил за выполнением и в конце работы снимал результаты, переводя их человекочитаемую форму.

Прогресс не стоял на месте и аппаратное обеспечение постепенно усложнялось, требуя всё больше действий для корректного запуска и остановки компьютера; нарушение порядка грозило поломкой дорогостоящей аппаратуры и просто потерей ценного машинного времени. Частично проблему решала постановка серийного производства компьютеров. Серии были мелкими, но всё же позволяли уменьшить затраты на использование за счёт типизации операций обслуживания и уменьшения стоимости подготовки специалистов.

С другой стороны, по мере всё большего распространения компьютеров, всё больше возникало типовых задач, решение которых требовало хоть и не одинаковых, но во многом похожих, а местами даже идентичных программ.

В итоге, естественным образом, стали появляться различные типовые куски программного кода, которые можно было использовать для решения большинства задач, не затрачивая время на их повторную разработку и отладку. Поначалу это были лишь отдельные подпрограммы — приведённые выше понятия интерфейсов ещё не различались, любой пользователь компьютера по определению был программистом. Но и этого оказалось мало.

Программы становились сложнее, и формулировать их в терминах конечной аппаратной платформы (то, что сейчас называется «программированием в машинных кодах») становилось также сложнее. Первым шагом к упрощению стало использование человекочитаемых мнемоник для программирования, которые хоть и не уменьшали количество операций, позволяли упростить (а в компьютерном мире «упрощение» практически всегда означает «улучшение», с точки зрения пользователя) процесс составления и редактирования программ. Такие изобретения мы сейчас называем языками (программирования) ассемблера. С тех пор они, к

слову, немного эволюционировали, но в любом случае, заострять внимание на этом в данном курсе мы не будем.

Следующим логическим шагом была замена одних человекочитаемых конструкций на другие, заметно более короткие. Так стали появляться высокоуровневые[1] языки программирования, в которых за одним оператором могут стоять сотни аппаратных инструкций. Интерпретаторы таких языков сначала распространялись в виде уже готовых наборов подпрограмм, а затем стали встраивать прямо в оборудование.

Параллельно со всем вышеперечисленным развивались функции по организации пакетного выполнения программ: после подготовки и загрузки программы, она сначала попадала в очередь на выполнение. При этом устройства ввода освобождались для следующей программы, избегая лишнего простоя как людей, так и оборудования. На достаточно продвинутых системах, вроде %%%, можно было управлять очередью: убирать отдельные задания или менять порядок их выполнения (%%% проверить). Такая очередь явилась предтечей файловой системы.

1 Изначально все языки программирования, не являвшиеся языками ассемблера, назывались высокоуровневыми. Однако ближе к 1990-ым появилось разделение на низкоуровневые (C, BASIC…) и высокоуровневые (Smalltalk, Ruby…) языки. Это деление в известной степени условно, так как многие языки «высокого уровня» позволяют использовать низкоуровневые конструкции (C++, Perl…).

Взаимодействие с оборудованием. Прерывания. Direct Memory Access

Компоненты внутри компьютера обмениваются информацией друг с другом посредством электрических сигналов, а точнее – выставления большего или меньшего напряжения на различных контактах и реагирования на этот факт. Мы не будем вдаваться в детали реализации, ограничившись сугубо теми возможными интерфейсами, которые оборудование предоставляет. В целом его можно разделить на три типа:

Регистры представляют собой малые области памяти (как правило, исчисляемые единицами или несколькими десятками бит). В зависимости от назначения регистра, они могут быть предназначены для только записи, только для чтения или для того и другого одновременно. Для доступа к регистрам обычно используется сопоставление их с адресами в оперативной памяти; таким образом, программа (ядро ОС) как будто обращается к обычной ячейке памяти, а по факту происходит обращение к оборудованию. Для этого зачастую задействуются специальные контроллеры ввода-вывода, и это уже выходит за рамки нашего курса.

Регистры – наиболее универсальный механизм взаимодействия с оборудованием, но он быстро становится неудобным при необходимости производить ввод-вывод на постоянной основе: так как скорость работы оборудования и центрального процессора (на котором исполняется код ядра ОС) обычно различается, приходится вводить различные задержки для операций чтения или записи. Как результат, мощности центрального процессора бесполезно простаивают, а пользовательский интерфейс, при его наличии, “заморожен”. Поэтому регистры используются чаще для настройки и управления режимами работы оборудования – операциями, которые происходят сравнительно редко. Также, из-за высокой себестоимости регистров, они не целесообразны для передачи больших объёмов данных.

Прерывания – механизм, посредством которого происходит прозрачное для выполняемого кода (пользовательского или ядерного) взаимодействие оборудования при возникновении требующей внимания ситуации, например: целочисленное переполнение, приход пакета данных из сети, нажатие пользователем кнопки и т.д. При обработке прерывания выполнение текущей программы, собственно, прерывается, и управление передаётся определённой процедуре-обработчику согласно таблице прерываний. В каком-то смысле это похоже на то, что посреди программы осуществляется вызов процедуры, не возвращающей какого-либо значения и не принимающей никаких параметров.

Обработчик прерывания обычно заканчивает свою работу максимально быстро, ограничиваясь только теми операциями, которые нельзя выполнить позднее. Например, в случае фиксирования нажатия клавиши, код клавиши (он ещё часто называется “скан-код”) записывается в системную очередь, которая обрабатывается уже в рамках обычной работы.

Если прерывание связано с обработкой исключительной ситуации, то в результате работы обработчика прерывания возможно изменение хода программы – например, “раскручивание” стека в поисках кода, который должен эту самую исключительную ситуацию обработать. В остальных случаях прерывания обычно полностью прозрачны для программы, их можно заметить только по косвенным признакам вроде скачка таймера – и то, на достаточно быстром аппаратном обеспечении это сделать крайне сложно. Хотя иногда и посредством такого временно́го анализа удаётся производить, например, взлом систем шифрования, путём последовательного вычисления бит ключа.

Третий, наиболее современный, механизм – это так называемый **прямой доступ к оперативной памяти** (direct memory access, DMA). Его суть заключается в том, что посредством того или иного протокола устройство и ядро ОС договариваются о том, что по такому-то адресу в оперативной памяти будет располагаться некоторая область заданного размера. В эту область устройство сможет что-то выводить, либо считывать из неё данные. Для этого, опять же, требуется помощь вспомогательных контроллеров (например, IOMMU), которыми требуется соответствующим образом управлять.

DMA позволяет значительно сократить накладные расходы на ввод-вывод. Например, размер сетевого пакета в современной локальной сети может составлять десятки килобайт, и приходить они могут в огромных количествах. В этом случае физический сетевой интерфейс (часто называемый сетевой картой) может записывать данные в оперативную память, после чего уведомлять прерыванием операционную систему о приходе новых сетевых пакетов; при этом за одно срабатывание обработчика прерываний могут быть обработаны сотни килобайт данных, безо всякого их копирования.

Текстовый (командный) интерфейс

Начнём с определения:

> *Интерфейс — набор средств взаимодействия с системой, позволяющий получать информацию о состоянии системы и/или изменять это состояние.*

Когда мы говорим об интерфейсе операционной системы, мы можем на самом деле говорить о нескольких принципиально разных сущностях:

- О пользовательском интерфейсе — он определяет, как пользователь может решать свои задачи посредством имеющихся программных инструментов, как из состава операционной системы, так и работающего в ней стороннего ПО.
- Об интерфейсе программирования приложений (API, Application Programming Interface) — он представляет собой набор спецификаций, которым должны удовлетворять исходные тексты ПО, чтобы оно корректно компилировалось под данную операционную систему.
- Об бинарном интерфейсе приложений (ABI, Application Binary Interface) — он определяет собой набор спецификаций, которым должно удовлетворять скомпилированное под данную операционную систему ПО для корректной работы под этой ОС.

API и ABI будут обсуждаться в других разделах %%%, здесь же мы будем говорить только о пользовательском интерфейсе.

Текстовые команды

Как уже говорилось в самом начале, взаимодействие с компьютером происходило в режиме «программа плюс данные плюс время = результат». Ошибки в программе приводили к необходимости её повторного ввода, вместе со всеми исходными данными; некоторых компьютерах эта задача могла частично упрощаться, за счёт, скажем, разделения линий ввода кода и данных, но принципиально задачу это не решало.

Прорывом идеологии стал диалог с компьютером: когда программа запрашивала данные, обрабатывала их, выдавала результат и снова ждала ввода. Таким образом программа могла работать неопределённо долгое время, обрабатывая произвольное (в рамках технических ограничений — например, те компьютеры весьма охотно ломались из-за обилия механических и/или ламповых компонентов) количество операций по желанию оператора (пользователя).

> *Примерно с этого момента началось разделение программных оболочек на предназначенные для ввода, редактирования и запуска программ, и на предназначенные для обслуживания всей системы. В то время как*

> *программистам требовались текстовые редакторы, компиляторы и скорость работы, операторы (сейчас их обычно называют системными администраторами) интересовались, в первую очередь, разделением доступа, удобством обслуживания и общей надёжностью систем, за которые они отвечали.*
>
> *С тех пор мало что изменилось в разделении труда. Однако сами программные средства изрядно эволюционировали. И в первую очередь это касается именно ввода и отображения информации.*

С появлением дисплеев и клавиатур стала возможной полноценного диалога, когда можно вводить программу сразу на языке высокого уровня, наблюдая её текст, а затем и результаты работы на экране. Поначалу речь шла только о текстовом представлении данных, однако позднее появилась возможность программно управлять любой точкой на экране (пикселем). Другими словами, компьютер смог показывать произвольные изображения. А изобретение Дугласа Энджелбарта под названием «мышь» стало началом эпохи, так как появилась возможность этим самым изображением управлять (почти) интуитивно понятным способом.

Так появились две наиболее популярные парадигмы пользовательского интерфейса: командный текстовый и графический, представляемые так называемыми (пользовательскими) оболочками — программами, реализующими собственно данный интерфейс.

Со временем появились и другие виды интерфейсов: голосовое управление, управление жестами, даже считывание мыслеобразов. Но пока что все эти интерфейсы так и остаются нишевыми решениями, не пригодными для универсального использования.

С графическим интерфейсом в любой современной стране можно столкнуться повсюду: от громадных телеэкранов на площадях до мобильных телефонов. Текстовый же, хотя и отошёл в тень из-за более высокого порога вхождения, отнюдь не вымер и вымирать не собирается. Почему? Это-то мы сейчас и разберём.

Прежде всего оговоримся, что нас интересует не любой текстовый интерфейс, а именно командный — то есть тот, в котором пользователь последовательно вводит некоторые команды, которые обрабатываются и результаты их работы (возможно) отображаются на экране. Итак, у нас появляются первые важные понятия:

Команда — минимальная грамматическая конструкция языка, понимаемого командной пользовательской оболочкой.

Пользовательская оболочка (шелл) — программа, реализующая диалог с пользователем. В ходе диалога пользователь вводит команды, а оболочка их выполняет и выдаёт результаты их работы.

В рамках курса мы будем изучать два распространённых семейства текстовых шеллов: поставляемый в комплекте с Windows CMD.EXE и стандартный для *nix шелл sh (Bourne Shell). На самом деле sh может быть представлен совершенно разными программами, но стандарт POSIX (см. раздел «Системные вызовы. API. Платформы POSIX и Win32», с. 20) подробно описывает требования к реализации, и мы будем, по возможности, придерживаться данного стандарта. Использование не стандартных, но широко распространённых особенностей будет каждый раз оговариваться в индивидуальном порядке.

Команда состоит обычно из следующих составных частей:

Название команды — имя встроенной команды оболочки, либо имя или путь к исполняемому файлу (программе).

Аргументы команды — текстовые строки, передаваемые оболочкой команде при запуске последней.

Модификаторы выполнения — дополнительные указания для оболочки о том, как следует выполнять данную команду.

Особняком стоит выставление параметров (переменных) шелла. Данные параметры могут использоваться просто как обычные переменные в языке программирования, но могут иметь и специальное значение, в зависимости от конкретного шелла.

Рассмотрим пример команды для CMD.EXE:

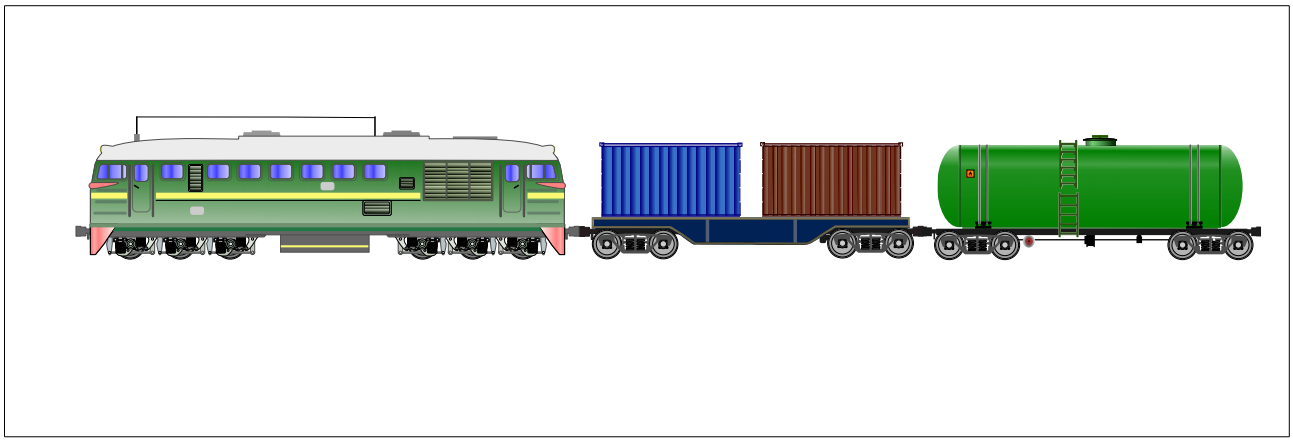
```
COPY /Y Diary.txt Diary_2015.txt
```

Имя команды здесь будет «COPY», а остальное — аргументами команды. Всего аргументов, как видно, три: в данном случае они просто отделяются друг от друга пробелами. Количество пробельных символов не принципиально, достаточно наличия между аргументами хотя бы одного.

Иногда аргумент сам по себе должен содержать пробелы. Например, мы хотим удалить файл «old archive.txt». В этом случае следует использовать кавычки:

```
DELETE "old archive.txt"
```

Кавычки можно использовать, в том числе, для задания пустого аргумента, то есть значением которого является пустая строка. Чтобы было понятнее, прибегнем к простой аналогии с поездом:



Представим, что этот поезд — команда шелла: локомотив — это название команды, платформ — аргументы. Теперь представим, что мы хотим посчитать число платформ, когда поезд проезжает мимо. Как мы можем это сделать? Самое простое — по пустому пространству между вагонами. Устанавливаем взгляд на середину высоты поезда, и получаем… три вагона вместо двух. Потому что на одной из платформ есть разрыв посередине, между контейнерами, и мы таким образом отсчитаем три пустых места. То же самое происходит в шелле, если написать вместо примера команды выше такую команду:

```
DELETE old archive.txt
```

Это будет воспринято как указание удалить два файла: «old» и «archive.txt». В лучшем случае вы получите две ошибки вида «файл не найден». В худшем случае такие файлы действительно найдутся и окажутся удалёнными. Это становится особенно важным, когда такие команды используются в скриптах, позволяющих не привилигерованным пользователям выполнять какие-либо задачи с повышенными привилегиями (см. «Разграничение доступа. Учётные записи пользователей и групп. Каталоги пользователей», с.25).

Рассмотрим ещё один пример команды:

```
printf "" >empty_file
```

Здесь название команды — «`printf`», и этой команде передан пустой аргумент — его создают в данном случае две двойные кавычки, идущие друг за другом и отделённые пробелами от остальных составляющих команды. Последняя часть команды — модификатор, указание для шелла по выполнению команды. Данный модификатор — «`>`» — указывает, что вывод команды следует *перенаправить* в файл «`empty_file`»; при этом, если файл ещё не существует, то он будет создан, а если существует — очищен. Таким образом, до запуска команды «printf» на выполнение шелл создаст (или очистит) файл «`empty_file`». Если в ходе выполнения данной операции шелл потерпит неудачу (например, файл отсутствует

и прав на его создание нет), команда запущена не будет; вместо этого шелл выведет сообщение об ошибке.

Зачем в команде выше понадобилось использовать пустой аргумент? — Дело в том, что «`printf`» требует указание как минимум одного аргумента (так же, как её прообраз — одноимённая функция в языке C). Если просто не написать в командной строке после имени команды ничего, кроме пробелов, то шелл не найдёт в такой командной строке аргументов как таковых и команда завершится с ошибкой, так как не будут удовлетворены её собственные требования к аргументам. Поэтому мы передаём одиночный пустой аргумент, в данном случае являющийся абсолютно корректным; команда просто ничего не выводит.

А зачем нужна такая команда, которая ничего не выводит? — Всё дело в модификаторе. Перед запуском команды шелл создаёт (или очищает) файл «`empty_file`». Таким образом, после выполнения команды можно быть уверенным, что в файловой системе в текущем каталоге появился файл «`empty_file`» без какого-либо содержимого. Вместо «`printf`» в данном случае можно использовать любую другую команду, которая гарантированно ничего не выводит на экран: «`echo -n`», «`true`» и т. д. — результат будет тот же самый.

Модификаторы в современных шеллах можно вставлять в любое место команды. Например, предыдущую команду без каких-либо изменений в её функциональности можно записать так:

```
printf >empty_file ""
```

Или даже так:

```
>empty_file printf ""
```

Исторически, тем не менее, принято ставить перенаправление ввода-вывода в конец команды — как показывает практика, это наиболее удобный в работе вариант.

Следует помнить, что порядок модификаторов *относительно друг друга* важен. Например, если указать два перенаправления в файл для одной команды, то ошибкой (для многих шеллов) это не будет являться:

```
echo hello >empty_file >another_file
```

Шелл при этом создаст/очистит оба указанных файла, но вывод команды отправится только во второй файл. Если нужно перенаправить вывод в несколько файлов одновременно, в *nix можно использовать команду «`tee`», передавая ей вывод предыдущей команды; для этого команды при помощи операции «`|`» объединяются в так называемый конвеер:

```
echo hello | tee destination_file1 >destination_file2
```

Здесь вывод команды «echo» отправляется не на экран, а на вход команды «tee». Последняя, как можно узнать из её документации, записывает всё, что приходит ей на вход, в каждый из указанных в качестве параметра файлов, а также отправляет дальше на свой стандартный выход; так как после команды «tee» нет конвеера, то в качестве стандартного выхода используется экран. В итоге и на экран, и в файл «destination_file1» отправится строчка «hello».

В таблице ниже перечислены некоторые модификаторы ввода-вывода, которыми можно воспользоваться в изучаемых нами шеллах:

мод-катор	команда	пример
>file	Перенаправляет обычный вывод команды в файл «file». Если файл не существует, он будет создан; если существует, его содежимое будет удалено.	echo Hi, M. >mail.txt
>>file	Перенаправляет обычный вывод команды в файл «file». Если файл не существует, он будет создан; если существует, вывод команды будет дописываться к уже имеющемуся содержимому файла.	date >>mail.txt
<file	Перенаправляет ввод команды из файла «file». Если файл не существует, шелл выдаст ошибку и команда запущена не будет.	
<<marker	Считывает все строки, начиная со следующей после данной команды и вплоть до строки, %% %перефразировать%%% имеющей значение «marker», и передаёт их на вход команды. Указанные строки обрабатываются аналогично содержимому двойных кавычек.	$ wc <<EOF In each eye Of the child — two eyes Of a puppy EOF 3 12 49
cmd1 | cmd2	Связывает обычный вывод команды «cmd1» со вводом команды «cmd2».	

Аргументы команд

Справочная информация

Вопреки расхожему мнению, в текстовых шеллах не так уж сложно разбираться. Одна из причин — единообразие интерфейса взаимодействия; другая

же — наличие документации; качество последней, правда, варьируется от системы к системе. Существуют стандартизированные механизмы получения справки, которые прежде всего делятся на встроенные в сами команды и внешнюю справку.

Встроенная в команду справка в Windows доступна обычно посредством указания аргумента `/?` у команды. В командах *nix обычно используется или `-?`, или `--help`, или оба. В любом случае обычным поведением команды, встретившей такой аргумент, является показ справочной информации о команде. Однако следует помнить, что, во-первых, у разработчиков ПО может быть и другое мнение о том, как должна команда реагировать на такие аргументы. А во-вторых, очевидно, не стоит запускать таким образом недоверенные программы, за поведение которых нет возможности поручиться: если такая программа подготовлена злоумышленником, то она сможет сделать то или иное чёрное дело с любым набором параметров командной строки.

В связи с вышеизложенным рекомендуется изначально использовать внешнюю справку. Для её получения в Windows существует команда HELP:

```
C:\> HELP COPY
```

Основным и фатальным недостатком команды HELP является то, что она описывает только встроенные команды CMD.EXE и программы, изначально поставляемые с Windows (и то не все). То есть, получить таким образом информацию о сторонней программе всё равно не получится, нужно использовать другие, не стандартизированные источники.

В *nix для получения справки существует команда `man`:

```
$ man cp
```

При этом справка является расширяемой и содержит страницы руководства[1] не только о программах, доступных из шелла, но и о форматах конфигурационных файлов, о функциях библиотек для различных языков программирования, системных вызовах и так далее. Если запрашиваемая страница руководства существует в нескольких категориях, то первой будет показана страница из более приоритетной категории. Приоритеты настраиваются в файле `/etc/man.conf`, справку по которому можно получить посредством команды:

```
$ man man.conf
```

Чтобы явно указать, из какой категории следует взять искомую страницу руководства, нужно написать номер раздела первым аргументом команды `man`, перед собственно названием страницы. Например, чтобы указать, что требуется

1 Название команды `man` происходит от англ. «manual» — «руководство».

страница с описанием функции `printf()`, а не, скажем, команды шелла `printf`, следует использовать следующую команду:

```
$ man 3 printf
```

Подробнее типичные номера разделов описаны в приложении «А» (с. 35).

Команда `man` позволяет производить поиск так же по краткому описанию команды, посредством опции `-k` (вывод будет отличаться в зависимости от ОС и установленного ПО):

```
$ man -k repeat
guniq(1) - report or omit repeated lines
gyes(1) - output a string repeatedly until killed
uniq(1) - report or filter out repeated lines in a file
SDL_EnableKeyRepeat(3) - Set keyboard repeat rate.
XAutoRepeatOff, XAutoRepeatOn, XBell, XChangeKeyboardControl,
XGetKeyboardControl, XKeyboardControl, XQueryKeymap(3) -
manipulate keyboard settings and keyboard control structure
XkbGetAutoRepeatRate(3) - Gets the current attributes of the
RepeatKeys control for a keyboard device
XkbGetDetectableAutoRepeat(3) - Determines whether or not the
server supports DetectableAutoRepeat
XkbSetAutoRepeatRate(3) - Sets the attributes of the
RepeatKeys control for a keyboard device
XkbSetDetectableAutoRepeat(3) - Sets DetectableAutoRepeat
idgen32, idgen32_init(9) - non-repeating ID generation
```

Отсылки к страницам руководства в документации по *nix обычно оформляются в виде `pagename(N)`, где «`pagename`» — название страницы, а «`N`» — номер категории. Иногда такую же запись используют для обозначения описываемых соответствующими страницами руководства сущностей, чтобы выделять их названия среди обычных слов и убирать неоднозначность при наличии одноимённых сущностей в нескольких категориях документации.

Полным синонимом «man -k» является команда apropos. Обе формы полностью эквивалентны и взаимозаменяемы.

Текстовые шеллы как языки программирования

Многие операции, осуществляемые в текстовых шеллах, легко поддаются автоматизации. В отличие от графического оконного интерфейса, где имеется множество непредсказуемых, параллельно изменяющихся параметров (расположение окон по трём осям, их размеры, активность и состояние отдельных элементов и т. д.), в текстовом командном интерфейсе события происходят последовательно[1], позволяя составлять устойчивые последовательности команд. Такие команды можно не только вводить непосредственно в командной строке, но и сохранять в текстовых файлах, которые потом можно запускать на выполнение, зачастую — так же, как обычные программы.

В Windows такие файлы имеют историческое расширение `.BAT` (от «batch» — пакет (команд)), или же `.CMD` (от «CMD.EXE»). Использование суффикса `.CMD` означает включение по умолчанию ряда дополнительных функций и особенностей `CMD.EXE`[2]. Так как `CMD.EXE` входит в состав всех современных версий Windows, рекомендуется использовать именно расширение `.CMD`. Тем не менее, в профессиональной среде скрипты для `CMD.EXE` по старинке часто называются «батниками».

В *nix нет требования определённого суффикса имени файла. Чтобы текстовый файл был интерпретирован шеллом, требуется поместить в начало файла специальную строчку вида:

```
#!/bin/sh
```

Где первые два символа («решётка» и восклицательный знак), называемые в англоязычной литературе «shebang», указывают ОС на необходимость запуска данного скрипта посредством команды, перечисленной далее. Имя файла будет передано отдельным аргументом к команде, которая может содержать и уже готовые аргументы. Например, для запуска awk-скрипта (содержащемся в файле с именем, скажем, `myawkprog`) потребуется задать имя скрипта через опцию `-f`:

```
#!/usr/bin/awk -f
```

Тогда для выполнения данного скрипта будет запущена следующая команда:

```
/usr/bin/awk -f myawkprog
```

1 Во многих современных шеллах есть возможность управлять запускаемыми командами: приостанавливать, прерывать и возобновлять их выполнение, отправлять команды выполняться в фоне и так далее. Подробнее см. XXX

2 Подробности см. во встроенной справке: `CMD.EXE /?`

Но вернёмся к шелл-скриптам. Управление доступом к файлам освещается далее, в разделе «Права доступа к файлам в *nix» (с. 25), а пока что для запуска шелл-скрипта можно использовать ту же технику, что описана выше, например:

```
sh myscript
```

В Windows можно также явно вызвать шелл, указав ему скрипт для запуска:

```
CMD /C myscript
```

На самом деле, помимо имени скрипта (точнее, пути к файлу со скриптом) можно указать любую команду. Также стоит отметить удобный для отладки альтернативный ключ `/K`: при использовании вместо `/C` он не завершит работу шелла вместе со скриптом, давая таким образом возможность ознакомиться с состоянием шелла после работы скрипта.

Однако при вызове одного скрипта из другого Windows и *nix ведут себя по-разному. В *nix это делается просто вызовом второго скрипта как обычной команды:

```
$ cat script1.sh
#!/bin/sh
echo -n calling 2nd script...
./script2.sh kva
echo " done"

$ cat script2.sh
echo " (this comes from 2nd script: $1)"

$ ./script1.sh
calling 2nd script... (this comes from 2nd script: kva) done
```

Однако в Windows аналогичная конструкция сработает неожиданным образом:

```
C:\> TYPE script1.cmd
@ECHO off
<NUL (SET /P t=calling 2nd script...)
script2.cmd kva
ECHO  done

C:\> TYPE script2.cmd
@ECHO off
<NUL ECHO (SET /P t=this comes from 2nd script: %1)

C:\> script1.cmd
calling 2nd script... (this comes from 2nd script: kva)
```

Как видно, script2.cmd был успешно вызван, однако выполнение script1.cmd на этом завершилось. Это — очередная унаследованная особенность CMD.EXE. Для вызова одного скрипта из другого требуется использовать команду CALL. Эту команду можно использовать и для запуска обычных программ, поэтому, если нет

уверенности, что запускаемая команда представлена обычной скомпилированной программой или является встроенной командой CMD.EXE, следует использовать команду CALL. Пример выше будет тогда выглядеть так:

```
C:\> TYPE script1.cmd
@ECHO off
<NUL (SET /P t=calling 2nd script...)
CALL script2.cmd kva
ECHO  done

C:\> TYPE script2.cmd
@ECHO off
<NUL ECHO (SET /P t=this comes from 2nd script: %1)

C:\> script1.cmd
calling 2nd script... (this comes from 2nd script: kva) done
```

Потоки выполнения. Процессы. Управление процессами

Системные вызовы. API. Платформы POSIX и Win32

Как уже говорилось выше, механизм обращения пользовательского ПО к функциям ядра ОС называется системными вызовами. В зависимости от конкретной аппаратной архитектуры системные вызовы могут инициироваться различными способами, сводящимися в основном к:

- программному вызову определённого прерывания, например, `int 80h` на архитектуре IA-32;
- специальной инструкции микропроцессора, например, `syscall` (код операции `0F 05`) на архитектуре AMD64.

Конкретная реализация обычно прячется за системными библиотеками, прежде всего — стандартной библиотекой языка C. Почему именно его? — потому что именно на C написаны критичные компоненты всех современных распространённых ОС общего (да и не только) назначения: Windows, OS X, Linux, BSD, Solaris… Конечно, специализированные ОС нередко пишутся на других языках: Oberon, Limbo и так далее. Однако именно использование C даёт, по нашему мнению, наиболее полное представление о работе программы и всей системы. Также стоит отметить, что большинство языков программирования поддерживает обращения из/в них посредством C-интерфейсов. Поэтому в дальнейшем все примеры компилируемых программ будут приводиться на этом языке.

Изначально язык C развивался безотрывно от ОС Unix (и её наследников). Неудивительно, что именно в Unix-подобных ОС язык C наиболее распространён. В частности, стандарт POSIX, определяющий собственно понятие «Unix-like», содержит много требований именно к C-интерфейсам и, фактически, дополняет стандарт на библиотеки языка и представляет собой набор обязательных и опциональных API.

> ***API*** *(Application Programming Interface, «интерфейс программирования приложений») — спецификация требований к интерфейсу библиотеки (или другого эквивалентного механизма), описывающая:*
>
> *названия используемых функций, структур, полей, констант и т. д.;*
>
> *передаваемые параметры и возвращаемые значения;*
>
> *протокол передачи параметров и возвращаемых значений;*
>
> *сторонние эффекты.*
>
> ***Сторонние эффекты*** *— воздействие на программы, ОС и другие объекты, не передаваемые в качестве параметров).*

Рассмотрим, как происходит процесс системного вызова на примере `chdir(2)` в ОС OpenBSD, платформа amd64:

- Программа вызывает функцию chdir(), определённую в заголовочном файле <unistd.h> и хранящуюся в составе libc.
- Данная является ассемблерной заглушкой (вывод команд сокращён)[1]:

```
$ objdump -x libc.a
<...>
0000000000000010  w    F .text  000000000000000d chdir
0000000000000010 g     F .text  000000000000000d .hidden
_libc_chdir
0000000000000000         *UND*  0000000000000000
_GLOBAL_OFFSET_TABLE_
0000000000000010 g     F .text  000000000000000d
_thread_sys_chdir
<...>
$ objdump -d chdir.o
chdir.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <_libc_chdir-0x10>:
   0:   48 8b 0d 00 00 00 00    mov    0(%rip),%rcx
   # ^ 7 <_libc_chdir-0x9>
   7:   ff e1                   jmpq   *%rcx
   9:   66                      data16
   a:   66                      data16
   b:   66                      data16
   c:   90                      nop
   d:   66                      data16
   e:   66                      data16
   f:   90                      nop
0000000000000010 <_libc_chdir>:
  10:   b8 0c 00 00 00          mov    $0xc,%eax
  15:   49 89 ca                mov    %rcx,%r10
  18:   0f 05                   syscall
  1a:   72 e4                   jb     0 <_libc_chdir-0x10>
  1c:   c3                      retq
```

Здесь видно, что в регистр `eax` (строго говоря, для amd64 это должен быть `rax`) записывается номер системного вызова (`0xc`), а в `r10` сохраняется так называемый static chain pointer — указатель на структуру, используемую в высокоуровневых языках программирования для ряда служебных целей. Остальные регистры, с аргументами для обработчика системного вызова, уже настроены к моменту вызова заглушки и дополнительной обработки не требуют.

---

1 Листинг представлен в формате GNU Assembler. Для тех, кто привык к нотации MASM/TASM: здесь у большинства операций естественный порядок операндов (сначала источник, потом назначение); регистры имеют префикс «%», а константы — «$»; косвенная адресация выглядит как «`смещение(%регистр)`».

После настройки регистров вызывается инструкция ЦП syscall, которая переключает режим работы ЦП в так называемое нулевое кольцо доступа (ring 0) и переводит выполнение на код ядра ОС. Ядро, в свою очередь, по завершении обработки запроса, вызывает функцию `sysret`, осуществляющую обратное переключение. Сам обработчик запроса выглядит в нашем случае так:

```
/* ARGSUSED */
int
sys_chdir(struct proc *p, void *v, register_t *retval)
{
        struct sys_chdir_args /* {
                syscallarg(const char *) path;
        } */ *uap = v;
        struct filedesc *fdp = p->p_fd;
        struct vnode *old_cdir;
        int error;
        struct nameidata nd;

        NDINIT(&nd, LOOKUP, FOLLOW | LOCKLEAF, UIO_USERSPACE,
            SCARG(uap, path), p);
        if ((error = change_dir(&nd, p)) != 0)
                return (error);
        old_cdir = fdp->fd_cdir;
        fdp->fd_cdir = nd.ni_vp;
        vrele(old_cdir);
        return (0);
}
```

Здесь видно ровно три аргумента, передаваемых обработчику: указатель на структуру контекста процесса, указатель на данные системного вызова и указатель на буфер для возвращаемого значения. Такой прототип используется для всех системных вызовов в данной ОС, поэтому не стоит удивляться, например, что `retval` не используется. Однако, откуда же берутся все эти параметры, ведь `syscall` их явно не оформляет в таком виде? Действительно, между запросом из пользовательского пространства и фактическим обработчиком системного вызова есть прослойка. В нашем случае она выглядит примерно так[1]:

```
void
syscall(struct trapframe *frame)
{
    caddr_t params;
    const struct sysent *callp;
    struct proc *p;
    int error;
    int nsys;
    size_t argsize, argoff;
    register_t code, args[9], rval[2], *argp;
```

1 См. файл src/sys/arch/amd64/amd64/trap.c в репозитории OpenBSD, ревизия №1.48.

```
    uvmexp.syscalls++;
    p = curproc;

    code = frame->tf_rax;
    callp = p->p_p->ps_emul->e_sysent;
    nsys = p->p_p->ps_emul->e_nsysent;
    argp = &args[0];
    argoff = 0;

    switch (code) {
    case SYS_syscall:
    case SYS___syscall:
         /*
          * Code is first argument, followed by actual args.
          */
         code = frame->tf_rdi;
         argp = &args[1];
         argoff = 1;
         break;
    default:
         break;
    }

    if (code < 0 || code >= nsys)
         callp += p->p_p->ps_emul->e_nosys;
    else
         callp += code;

    argsize = (callp->sy_argsize >> 3) + argoff;
    if (argsize) {
         switch (MIN(argsize, 6)) {
         case 6:
              args[5] = frame->tf_r9;
         case 5:
              args[4] = frame->tf_r8;
         case 4:
              args[3] = frame->tf_r10;
         case 3:
              args[2] = frame->tf_rdx;
         case 2:
              args[1] = frame->tf_rsi;
         case 1:
              args[0] = frame->tf_rdi;
              break;
         default:
              panic("impossible syscall argsize");
         }
         if (argsize > 6) {
              argsize -= 6;
              params = (caddr_t)frame->tf_rsp +
                  sizeof(register_t);
              if ((error = copyin(params, &args[6],
                  argsize << 3)))
```

```
                    goto bad;
          }
     }

     rval[0] = 0;
     rval[1] = frame->tf_rdx;

     error = mi_syscall(p, code, callp, argp, rval);

     switch (error) {
     case 0:
          frame->tf_rax = rval[0];
          frame->tf_rdx = rval[1];
          frame->tf_rflags &= ~PSL_C;   /* carry bit */
          break;
     case ERESTART:
          /* Back up over the syscall instruction (2 bytes) */
          frame->tf_rip -= 2;
          break;
     case EJUSTRETURN:
          /* nothing to do */
          break;
     default:
     bad:
          frame->tf_rax = error;
          frame->tf_rflags |= PSL_C;    /* carry bit */
          break;
     }

     mi_syscall_return(p, code, error, rval);
}
```

Уже из прототипа функции видно, что системный вызов на данной архитектуре — частный случай исключительной ситуации (trap). Исключительные ситуации возникают в системе по разным случаям: ошибка доступа к памяти, ошибка при вычислениях с плавающей запятой, отсутствие выравнивания при доступе к памяти (на архитектурах, где это критично) и так далее. Благодаря механизму обработки таких ситуаций ядро ОС может предпринять какие-либо действия, или даже предложить программе самой справиться с данной ошибкой: например, программа-отладчик будет заинтересована в перехвате ошибок вида «доступ к памяти за выделенными программе пределами».

Данная функция вызывается из низкоуровневого кода

Разграничение доступа. Учётные записи пользователей и групп. Каталоги пользователей

Права доступа к файлам в *nix

Современная файловая система. Журналирование. Снимки состояния и резервное копирование

Прежде, чем говорить о файловой системе, попробуем определиться с тем, что такое файл. Классическое определение звучит так:

> *Файл — уникально именованная область данных на диске (в ПЗУ).*

Однако в современном мире с данными определениями есть некоторые сложности:

Прежде всего, насколько уникально может быть именован файл? Если у нас несколько дисков, то на каждом из них мы можем иметь файлы с одним и тем же названием. Более того, даже на одном диске мы можем иметь файлы с одинаковыми именами, если они находятся в разных каталогах. Впрочем, последнее легко объясняется введением понятия «полного пути к файлу», которое включает в себя каталог, в котором находится файл, и все вышестоящие каталоги, вплоть до корневого. Но мы забежали вперёд, о каталогах говорить ещё рано.

Вернёмся к определению файла. Если мы говорим о файле, как области данных именно на диске — как можно называть диском, скажем, карту флеш-памяти, в которой нет ни одной движущейся части? Более корректно будет говорить о носителе информации.

Если же мы говорим о ПЗУ в целом — как быть с теми случаями, когда носитель информации является виртуальной сущностью? Например, часть оперативной памяти может быть определена именно под хранение файлов, как на обычном диске. Конечно, такой «диск» просуществует не дольше, чем потребуется электронам рассеяться после выключения питания, но пользоваться им можно будет точно так же. Или, скажем, какой-либо файл может быть представлен как носитель информации. А ведь ещё бывают файлы, содержание которых вообще нигде не записывается: при чтении их содержимое генерируется на лету, а при записи — если таковая разрешена — обрабатывается после закрытия файла…

Понятие «носитель информации» использовать мы не можем, так как оно относится к физическому объекту (например, тому же жёсткому диску). А жёсткий диск, как известно, может быть разделён на несколько, как бы неожиданно это ни прозвучало, разделов. И в каждом таком разделе данные могут храниться независимым образом.[1]

---

1 Здесь может возникнуть путаница из-за того, что семействах операционных систем DOS и Windows слово «диск» обозначает обычно именно раздел носителя информации, а не весь носитель. Поэтому, чтобы избежать данной путаницы, мы постараемся не использовать понятие «диск» без чётко обозначенного контекста.

Может, тогда использовать «раздел»? Однако, далеко не все носители данных могут быть разделены.[1] Можно, конечно, считать, что на носителе информации всегда присутствует хотя бы один раздел, но на определённом уровне абстракции это становится некорректным: когда мы говорим о наличии раздела мы подразумеваем наличие где-то на самом носителе таблицы разделов. Эта информация, хотя может и не представляться сама в виде файлов, занимает какое-то место на диске — так же, как оглавление в книге.

Итак, мы приходим к тому, что сначала нужно определить саму файловую систему. Мы будем пользоваться следующим определением:

> *Файловая система — виртуальная сущность, упорядочивающая данные с использованием уникальных, как правило — образующих иерархию, идентификаторов, в определённых границах физического носителя или носителей, а также предоставляющая доступ пользовательским приложениям к этим данным посредством стандартизированных механизмов операционной системы.*

Файловые системы, в зависимости назначения, могут обладать сильно различающимися характеристиками:

- Быть доступными только для чтения, или также для записи;
- Иметь технологии восстановления после сбоев;
- Иметь поддержку транзакций или моментальных снимков (snapshots);
- Поддержка того или иного механизма разграничения доступа к файлам;
- Поддержка жёстких ссылок (один и тот же файл под разными местами и/или в разных каталогах в пределах данной ФС);
- Поддержка символических ссылок (см. ниже);
- Возможность сохранять произвольные метаданные, или даже иметь полноценные альтернативные контейнеры данных под одним и тем же именем файла (например, т. н. потоки (streams) в NTFS);
- и так далее.

Все вышеуказанные возможности будут в той или иной степени рассмотрены ниже.

1 В отсутствие собственной таблицы разметки диска в операционной системе, вроде BSD disklabel; подробнее об этом ниже.

Файловые операции. Файловые дескрипторы

Представьте себе типичный парк аттракционов. Чтобы попасть в этот парк, требуется купить билет, после чего можно пользоваться всеми аттракционами вплоть до выхода из парка (или его закрытия). Однако, если вы выйдете из парка, а затем захотите в тот же день ещё раз в него зайти, вам потребуется покупать билет снова.

Точно так же происходит работа с файлами. Чтобы произвести любую операцию с содержимым файла, его требуется вначале «открыть». В результате выполнения данной операции возвращается так называемый дескриптор — некая сущность, однозначно идентифицирующая открытый файл. При этом, так же, как можно купить несколько билетов и пользоваться любым из них в парке аттракционов, можно открыть несколько дескрипторов для одного и того же файла в одной и той же файловой системе.[1]

В ОС *nix файловые дескрипторы представляются неотрицательными целыми числами. При этом первые три дескриптора имеют заранее определённое назначение:

Дескриптор №0 представляет собой стандартный ввод (standard input, stdin) программы. Многие программы, обрабатывающие данные в поточном режиме, используют стандартный ввод при отсутствии явного списка открываемых файлов: `cat`, `sh`, `awk`...

Дескриптор №1 представляет собой стандартный вывод (standard output, stdout) программы. Именно в этот дескриптор происходит вывод функций `printf()`, `puts()` и т. д.

Дескриптор №2 так же представляет собой вывод, но уже — диагностических сообщений (standard error output, stderr). Он предназначен для вывода программой диагностических сообщений, приглашений для пользователя и других данных, которые не должны попадать в стандартный вывод. Например, команда dump может осуществляет вывод содержимого дампа файловой системы в stdout (который может быть перенаправлен в файл или другую программу), а stderr при этом используется для вывода сообщений о текущем состоянии снятия резервной копии. Таким образом рабочие данные, передаваемые по stdout, не искажаются.

Помимо предназначения, stderr во многих ОС отличается тем, что, в отличие от stdout, не буферизуется. Буферизация делается стандартной библиотекой (языка

1 В некоторых системах, прежде всего в Windows, присутствует возможность открыть файл в эксклюзивном режиме: системе гарантирует, что более никакая программа, в том числе та же самая, не сможет открыть файл ещё раз; а если файл уже открыт, то попытка открыть его в эксклюзивном режиме провалится. До недавнего времени открытие в эксклюзивном режиме было поведением по умолчанию.

C) для того, чтобы избежать избыточных системных вызовов: если программа, например, записывает данные (в stdout) по несколько байт, каждый раз вызывать write()/WriteFile() довольно накладно. Поэтому данные для записи сначала накапливаются в буфере до определённого минимального размера (или, скажем, до появления в отправляемых данных символа конца строки — такой подход называется построчной буферизацией, line buffering), а потом выводятся все вместе. Типичными буферизуемыми операциями в C являются `printf()`, `puts()` и так далее — по сути, все функции вывода, объявленные в `<stdio.h>`.

При открытии файла (или другой файловой сущности, например, сокета) «открывается» файловый дескриптор с минимально доступным номером. Например, если программой открыты дескрипторы №№ 0, 1, 2, 3 и 6, то последующие открываемые файлы получат дескрипторы №№ 4, 5, 7, 8, 9 и так далее. Вместе с операциями `dup()` и, особенно, `dup2()` такой механизм позволяет получить при необходимости строго определённый дескриптор — в частности, один из специальных 0…2, описанных выше. В частности, именно таким образом работает перенаправление ввода-вывода в шелле (пример возможной реализации):

```
int
redirect(int dstfd, const char *path, int flags, mode_t m) {
    int fd, error;

    fd = open(path, flags, m);
    if (fd == -1)
        return -1;
    error = (dup2(fd, dstfd) == -1);
    close(fd);
    return error ? -1 : 0;
}
```

Пример команды, перенаправляющей конкретные дескрипторы:

```
ls * >/tmp/listing 2>/tmp/errors
```

В результате в файле `/tmp/listing` окажется вывод команды `ls`, за исключением сообщений об ошибках (если таковые будут), которые попадут уже в файл `/tmp/errors`. На экран, соответственно, не попадёт ничего.

Можно использовать номера дескрипторов не только для указания назначения (в левой части модификатора перенаправления ввода-вывода), но и для указания источника (в правой части модификатора перенаправления ввода-вывода). Для этого используется спецсимвол амперсанд («&»):

```
cat /var/log/*.log >/tmp/combined.log 2>&1
```

Данная команда объединит содержимое всех файлов с суффиксом «.log»в файл /tmp/combined.log, а также направит в этот же файл все сообщения об ошибках (скажем, про нехватку прав доступа).

В Windows файловые дескрипторы (а также дескрипторы многих других подсистем) представляются непрозрачным типом HANDLE. Из-за непрозрачности (и отсутствия аналогов функций dup*) механизмы, аналогичные описанному выше, не работают. Тем не менее, стандартный шелл Windows поддерживает перенаправление стандартных файловых дескрипторов посредством параметров, передаваемых функции CreateFile()[1]:

```
DIR >%TEMP%\Listing.txt 2>%TEMP%\Errors.txt
```

и:

```
TYPE *.log >%TEMP%\Combined.log 2>&1
```

Следует помнить, что знак «>» или «<» означает только режим открытия файла. Из этого, во-первых, следует, что вы можете написать «2<foo.txt», и это будет в каком-то смысле работать — но в общем случае последствия могут быть не предсказуемыми.

1 Подробнее см. MSDN: Creating a Child Process with Redirected Input and Output — https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms682499(v=vs.85).aspx

Сетевой стек ОС. Основы маршрутизации в сетях TCP/IP. Сокеты

Модель OSI. TCP/IP-стек. Модель TCP/IP

Основы маршрутизации

Сокеты. Неблокирующийся ввод-вывод

Устройство исполняемого файла. Динамические (разделяемые) библиотеки. ABI

Сборка и установка ПО в современных ОС

Заключительная. Лицензирование ПО. Открытое ПО, его разновидности

Приложения

A. Категории справки команды man

Номера категорий могут отличаться в разных ОС, поэтому данная таблица носит скорее вспомогательный характер, чем справочный. Для получения точных сведений следует смотреть описание команды `man` и иже с ней в конкретной ОС.

Номер	Описание
1	Команды шелла, предназначенные для обычной пользовательской деятельности (`*/bin`)
2	Системные вызовы (см. с. 20)
3	Функции библиотек языка C (в том числе, стандартной библиотеки)
3p, 3perl	Функции библиотек языка Perl (в том числе описание языка)
4	Драйвера устройств и другие низкоуровневые программные интерфейсы, доступные из пользовательского режима
5	Описание форматов файлов, в первую очередь — конфигурационных
6	Игры
7	Раздел для тех страниц, которые не подходят в другие разделы; в частности, здесь обычно размещаются страницы с описаниями сложных программных комплексов, вроде `samba(7)`.
8	Команды шелла, предназначенные в первую очередь для администрирования ОС (`*/sbin`)
9	Справка по ядру ОС
0	Псевдораздел, используется в некоторых ОС для обозначения заранее отформатированных для показа страниц