

# КОНФИГУРАЦИОННОЕ УПРАВЛЕНИЕ

П.Н. Советов

Москва – 2021

# Оглавление

<b>1. Введение</b>	<b>1</b>
1.1. О чем этот курс . . . . .	1
1.2. Основные определения . . . . .	2
<b>2. Работа в командной строке</b>	<b>8</b>
2.1. Командный интерпретатор . . . . .	10
2.2. Инструменты <code>grep</code> , <code>sed</code> и <code>awk</code> . . . . .	20
<b>3. Менеджеры пакетов</b>	<b>23</b>
3.1. Нумерация версий ПО . . . . .	23
3.2. Управление пакетами . . . . .	25
<b>4. Конфигурационные языки</b>	<b>28</b>
4.1. Виды конфигурационных языков . . . . .	28
4.2. Представление синтаксиса языка с помощью БНФ . . . . .	29
4.3. Простые форматы описания конфигурации . . . . .	29
4.4. Языки общего назначения . . . . .	35
4.5. Программируемые DSL для задач конфигурирования . . . . .	36
<b>5. Системы автоматизации сборки</b>	<b>38</b>
5.1. Топологическая сортировка графа зависимостей . . . . .	42
5.2. Система сборки <code>make</code> . . . . .	44
<b>6. Системы контроля версий</b>	<b>47</b>
6.1. История развития систем контроля версий . . . . .	54
6.2. Система <code>Git</code> . . . . .	55
6.3. Модель данных <code>Git</code> . . . . .	57
<b>7. Документация как код</b>	<b>61</b>
7.1. Языки разметки . . . . .	61

7.2. Грамотное программирование . . . . .	64
7.3. Markdown и Pandoc . . . . .	66
7.4. Языки описания диаграмм . . . . .	67
7.5. Генераторы документации на основе исходных текстов . . . . .	70
<b>8. Виртуализация</b>	<b>72</b>
8.1. Что такое виртуализация . . . . .	72
8.2. Языковые виртуальные машины . . . . .	73
8.3. Виртуализация вычислительной системы . . . . .	76
8.4. Виртуализация приложения . . . . .	79
8.5. Виртуализация уровня ОС . . . . .	79



# 1. Введение

## 1.1. О чем этот курс

Программная инженерия – это то, что происходит с программированием при добавлении времени и других программистов”

(Russ Cox)

Отслеживание и управление изменениями в ПО – важная задача, которая возникает даже при работе над небольшими индивидуальными проектами. В этом курсе предлагается практико-ориентированный взгляд на конфигурационное управление. Изучаемые темы:

1. командная строка,
2. менеджеры пакетов,
3. конфигурационные языки,
4. системы автоматизации сборки,
5. системы контроля версий,
6. документация как код,
7. виртуализация.

Изначально управление конфигурацией применялось не в программировании. Под конфигурацией понимался состав деталей конечного продукта и «взаимное расположение частей» физического изделия. Таким образом, конфигурацией можно управлять, контролируя документы, описывающие конечный продукт, требования к нему, всю его проектную и технологическую документацию.

В связи с высокой динамичностью сферы разработки ПО, в ней конфигурационное управление особенно полезно. К процедурам можно отнести:

- управление версиями проекта,
- управление сборками проекта,
- контроль требований проекта,

- контроль документации
- и т.д.

Степень формальности выполнения данных процедур зависит от размеров проекта, и при правильном подходе данная концепция может быть очень полезна.

## 1.2. Основные определения

**Конфигурационное управление** – дисциплина идентификации компонентов системы, определения функциональных и физических характеристик аппаратного и программного обеспечения для проведения контроля внесения изменений и отслеживания конфигурации на протяжении ЖЦ, см. рис. 1. Это управление соответствует одному из вспомогательных процессов ЖЦ (ISO/IEC 12207), выполняется техническим и административным руководством проекта и заключается в контроле указанных характеристик конфигурации системы и их изменении; составления отчета о внесенных изменениях в конфигурацию и статус их реализации; проверки соответствия внесенных изменений заданным требованиям.

**Конфигурация системы** – состав функций, программных и физических характеристик программ или их комбинаций, аппаратного обеспечения, обозначенные в технической документации системы и реализованные в продукте.

**Элемент программной конфигурации** – фрагмент программного обеспечения, вовлеченный в процесс конфигурационного управления и рассматриваемый как одна (атомарная) сущность.

К элементам конфигурации относятся такие объекты, как:

- рабочие документы;
- файлы исходных кодов;
- файлы ресурсов;
- файлы, создаваемые в результате сборки (исполняемые файлы, библиотеки и так далее);
- инструменты, используемые для разработки (их мы тоже должны учитывать для стандартизации и упрощения взаимодействия в команде);

**Версия** – это состояние элемента программной конфигурации, которое может быть восстановлено в любой момент времени независимо от истории изменения.



Рис. 1. Цели и задачи конфигурационного управления

Элементы системы, их версии и конфигурации показаны на рис. 2.

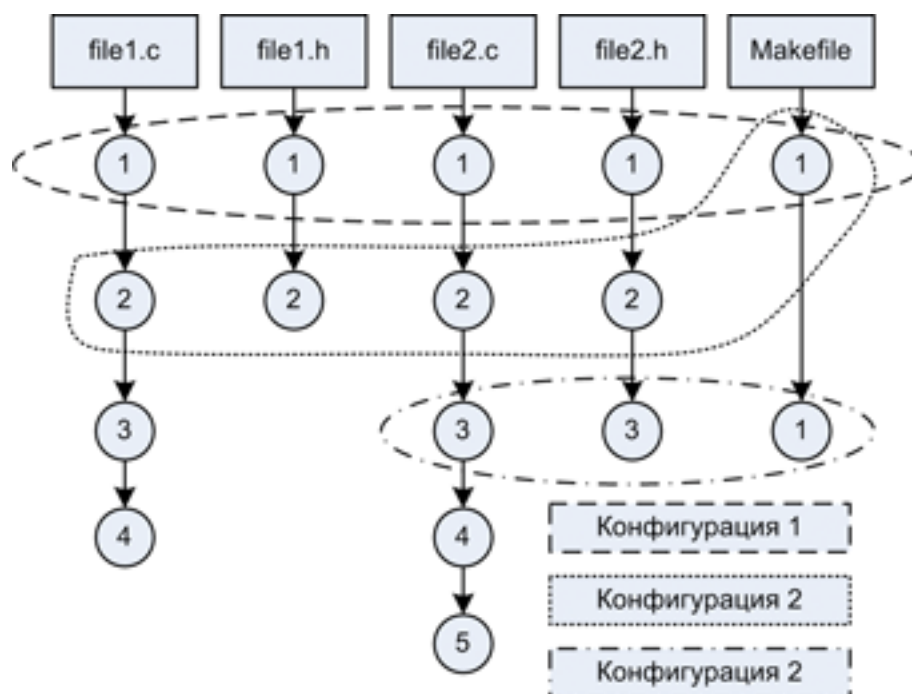


Рис. 2. Элементы, их версии и срезы-конфигурации

Конфигурация ПО включает набор функциональных и физических характеристик ПО, заданных в технической документации и достигнутых в готовом продукте. Т.е это сочетание разных элементов продукта вместе с заданными процедурами сборки и отвечающие определенному назначению. Элемент конфигурации – график разработки, проектная документация, исходный и исполняемый код, библиотека компонентов, инструкции по установке системы и др.

Область знаний «Управление конфигурацией ПО» состоит из следующих разделов:

- управление процессом конфигурацией (Management of SMC Process),
- идентификация конфигурации ПО (Software Configuration Identification),
- контроль конфигурации ПО (Software Configuration Control),
- учет статуса конфигурации ПО (Software Configuration Status Accounting),
- аудит конфигурации ПО (Software Configuration Auditing),
- управление релизами (версиями) ПО и доставкой (Software Release Management and Delivery).

Управление процессом конфигурации – это деятельность по контролю эволюции и целостности продукта при идентификации, контроле изменений



и обеспечении отчетности информации, касающейся конфигурации. Включает:

- систематическое отслеживание вносимых изменений в отдельные составные части конфигурации и проведение аудита изменений и автоматизированного контроля за внесением изменений в конфигурацию системы или ПО;
- поддержка целостности конфигурации, ее аудит и обеспечение внесения изменений в один объект конфигурации, а также в связанный с ним другой объект;
- ревизия конфигурации на предмет проверки разработки необходимых программных или аппаратных элементов и согласованности версии конфигурации с требованиями;
- трассировка изменений в конфигурацию на этапах сопровождения и эксплуатации ПО.

**Идентификация конфигурации ПО** проводится путем выбора элемента конфигурации ПО и документирования его функциональных и физических характеристик, а также оформления технической документация на элементы конфигурации ПО.

**Контроль конфигурации ПО** состоит в проведении работ по координации, утверждению или отбрасыванию реализованных изменений в элементы конфигурации после формальной ее идентификации, а также оценке результатов.

**Учет статуса конфигурации ПО** проводится в виде комплекса мероприятий для определения уровня изменений в конфигурацию, аудита конфигурации в виде комплекса мероприятий по проверке правильности внесения изменений в конфигурацию ПО. Информация и количественные показатели накапливается в соответствующей БД и используются при управлении конфигурацией, составлении отчетности, оценке качества и выполнении других процессов ЖЦ.

**Аудит конфигурации** – это деятельность, которая выполняется для оценки продукта и процессов на соответствие стандартам, инструкциям, планам и процедурам. Аудит определяет степень удовлетворения элемента конфигурации заданным функциональным и физическим характеристикам системы. Различают функциональный и физический аудит конфигурации, который завершается фиксацией базовой линии (базиса) продукта. Сборка ПО – объединение корректных элементов ПО и конфигурационных данных в единую исполняемую программу.

Процедуру сборки проекта часто автоматизируют, то есть выполняют не из среды разработки, а из специального скрипта – **build-скрипта**. Этот скрипт используется тогда, когда разработчику требуется полная сборка всего проекта. А также он используется в процедуре непрерывной интеграции (continuous integration) – то есть регулярной сборке всего проекта (как правило – каждую ночь). Во многих случаях процедура непрерывной интеграции включает в себя и регрессионное тестирование, и часто – создание инсталляционных пакетов. Общая схема автоматизированной сборки представлена на рис. 3.



Рис. 3. Схема автоматизированной сборки ПО

**Стабилизация конфигурации** – это процесс получения новой конфигурации из имеющихся промежуточных конфигураций. Для этого процесса также используются также термины «выпуск» или «релиз». Результат стабилизации также может быть назван, в свою очередь, релизом или выпуском.

**Управление релизами (версиями) ПО** это: отслеживание имеющейся версии элемента конфигурации; сборка компонентов; создание новых версий системы на основе существующей путем внесения изменений в конфигурацию; согласование версии продукта с требованиями и проведенными изменениями на этапах ЖЦ; обеспечение оперативного доступа к информации относительно элементов конфигурации и системы, к которым они относятся. Управление выпуском охватывает идентификацию, упаковку и передачу элементов продукта и документации заказчику.

**Базис (baseline)** – формально обозначенный набор элементов ПО, зафиксированный на этапах ЖЦ ПО. Это конфигурация, выбранная и закреплённая на любом этапе жизненного цикла разработки как основа для дальнейшей работы.

На рис. 4 показан пример появления конфигураций во времени.

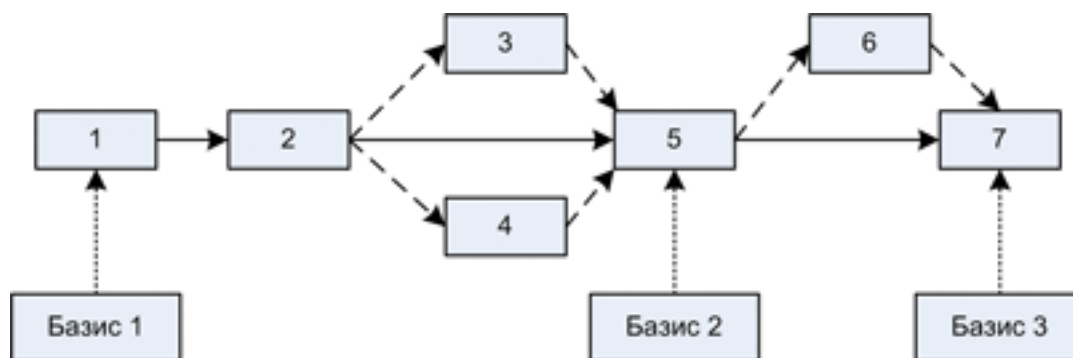


Рис. 4. Пример появления конфигураций во времени

Начальное состояние проекта – конфигурация 1. Она же является первым базисом, от которого будет идти дальнейшая разработка. Предположим, проект на начальной стадии. Через какое-то время появляется обновленная конфигурация 2. Разработка только началась и был выпущен релиз, чтобы дать команде какую-то основу для дальнейшей работы. В ходе проверки выяснилось, что базой для работы этот выпуск служить не может – есть непонятные и противоречивые места.

Для их устранения группы разработки делают доработки. В результате них появляются конфигурации 3 и 4 – обе они разработаны на основе 2, но друг с другом они не согласуются, поскольку не включают изменения друг от друга. СМ-инженер создает конфигурацию 5, сделанную на основе 2, 3 и 4. После проверки менеджмент объявляет конфигурацию базовой. По этому сигналу СМ-команда выпускает этот релиз как официальную базовую конфигурацию и разработчики берут уже ее за основу. Далее история повторяется, группа разработки вносит изменения – появляется конфигурация 6. Ее, в свою очередь, интегрирует СМ-инженер и она получает номер 7. Он также становится официальной базой для разработки.

## 2. Работа в командной строке

Операционная система UNIX была разработана в далеком 1969 году. UNIX изначально являлась операционной системой в первую очередь для разработчиков, которым удобнее всего автоматизировать свои действия с помощью командной строки. Сама по себе командная строка еще древнее UNIX. Командная строка на экране монитора – имитация работы с телетайпом. Телетайп, в свою очередь, является электромеханической печатной машиной, которую можно подключить к компьютеру. Пользователь набирает текст, который печатается на рулоне бумаги. Компьютер печатает пользователю свои ответы. Удивительно, но такой, казалось бы, устаревший способ общения с компьютером все еще активно используется. Более того, многие задачи очень трудно решить без командной строки! Это касается, в частности, работы с системой контроля версий Git, с системой автоматизации сборки Make, с системой контейнеризации Docker и многими другими популярными сегодня программами. Командная строка в духе UNIX имеется в MacOS, Linux и Windows (WSL, Powershell). Стоит вспомнить и многочисленные фильмы о «хакерах» – если герой фильма решает за компьютером какие-то нетривиальные задачи, то, обычно, зрителю демонстрируется именно командная строка.

Основная суть решений, принятых при использовании командной строки, сводится к положениям «философии UNIX» (Дуг Макилрой), которые можно выразить следующими пунктами:

- Пусть каждая программа решает хорошо одну задачу. Для новых задач создавайте новые программы, а не усложняйте старые новыми «возможностями».
- Предполагайте, что вывод каждой программы может стать входом другой, еще неизвестной программы. Не загромождайте вывод посторонней информацией. Избегайте строго выравненных столбчатых форма-

тов и двоичных входных форматов. Не настаивайте на интерактивном вводе.

- Проектируйте и разрабатывайте ПО, даже операционные системы, таким образом, чтобы его можно было опробовать уже на ранних этапах, в идеале в течение недель. Не бойтесь выбрасывать неудачно реализованные части и пересоздавать их.
- Вместо ручного труда используйте инструменты для облегчения задач разработки, даже если придется отвлечься на создание этих инструментов, а сами инструменты могут впоследствии больше не понадобиться.

Далее будет рассматриваться современный вариант UNIX, популярная ОС Linux, разработанная в 1991 году Линусом Торвальдсом (в те времена – студентом финского университета).

При работе с командной строкой необходимо учитывать структуру файловой системы. В Linux она имеет следующий вид:

```
localhost:~# tree -d -L 1 /
/
├── bin
├── dev
├── etc
├── home
├── lib
├── media
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin
├── srv
├── sys
├── tmp
├── usr
└── var
```

Имя / является корнем (root) файловой системы. Внутри корня расположены следующие, наиболее значимые каталоги:

- bin. Исполняемые файлы системных утилит.
- dev. Устройства, к которым в Linux возможен доступ, как к файлам.

- etc. Системные конфигурационные файлы.
- home. Домашние каталоги пользователей.
- lib. Системные библиотеки.
- media. Подключаемые съемные диски: USB, CD-ROM и так далее.
- mnt. Подключаемые разделы.

## 2.1. Командный интерпретатор

За поддержку работы в командной строке отвечает специальная программа – интерпретатор оболочки ОС (shell). В случае Linux таким интерпретатором обычно является Bash, архитектура которого приведена на рис. 5. Интерпретатор осуществляет следующие действия:

1. Принимает строку от пользователя.
2. Разбирает эту строку и переводит во внутренний формат.
3. Осуществляет подстановку для различных специальных символов и имен.
4. Исполняет команду пользователя.
5. Выдает код выполнения.

Ниже показан пример сеанса работы в Bash:

```
localhost:~# pwd
/root
localhost:~# ls -l
total 16
-rw-r--r--    1 root    root          114 Jul  5  2020 bench.py
-rw-r--r--    1 root    root           76 Jul  3  2020 hello.c
-rw-r--r--    1 root    root          22 Jun 26  2020 hello.js
-rw-r--r--    1 root    root         151 Jul  5  2020
↳  readme.txt
localhost:~# echo 'new file' > new_file.txt
localhost:~# cat new_file.txt
new file
localhost:~# mkdir new_dir
localhost:~# cp new_file.txt new_dir/
localhost:~# rm new_file.txt
localhost:~# ls -l
total 20
-rw-r--r--    1 root    root          114 Jul  5  2020 bench.py
-rw-r--r--    1 root    root           76 Jul  3  2020 hello.c
```

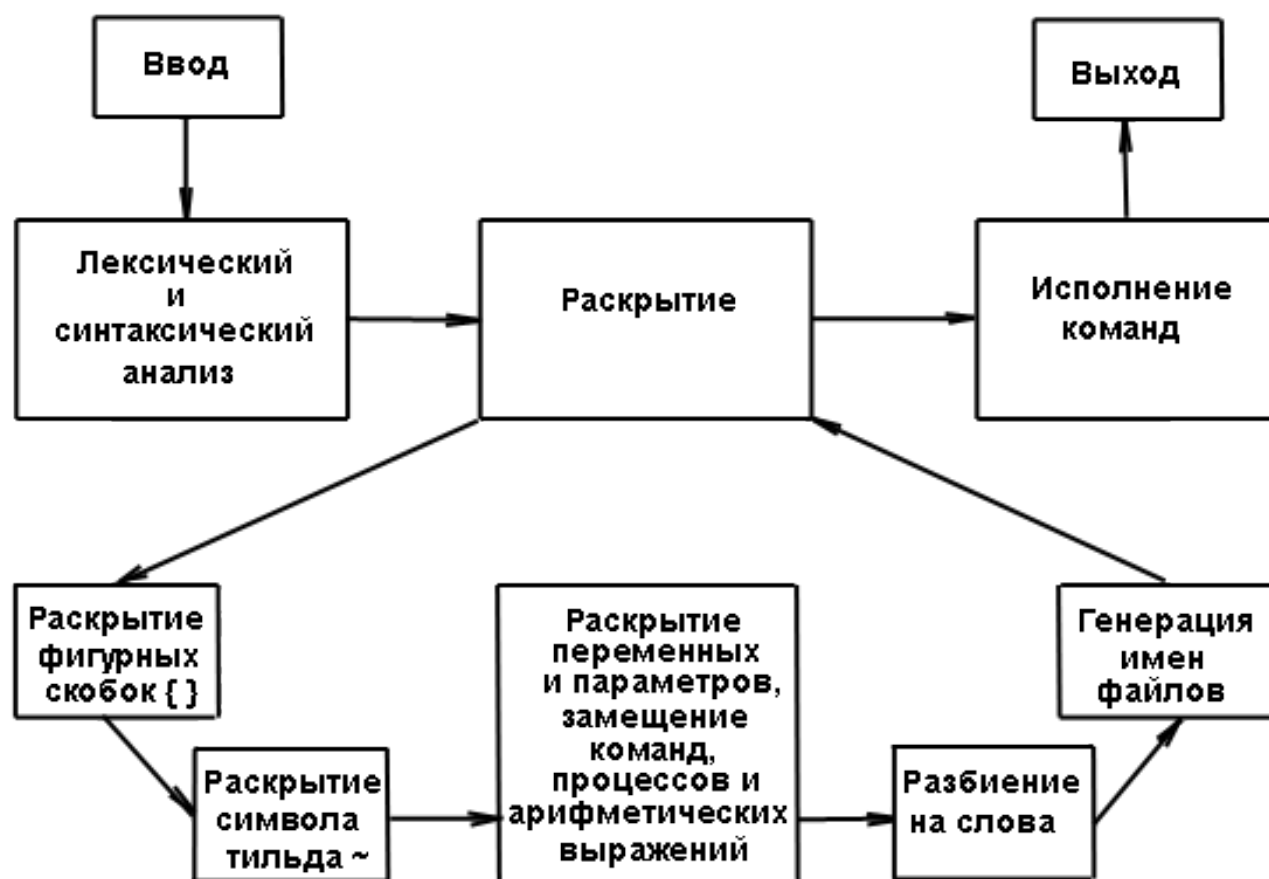


Рис. 5. Архитектура интерпретатора Bash

```

-rw-r--r--      1 root      root      22 Jun 26  2020 hello.js
drwxr-xr-x      2 root      root      66 Nov  4 17:16 new_dir
-rw-r--r--      1 root      root     151 Jul  5  2020
↪  readme.txt
localhost:~# ls -l new_dir/
total 4
-rw-r--r--      1 root      root      9 Nov  4 17:16
↪  new_file.txt

```

Обратите внимание на использование в приведенном сеансе команд, упрощенное описание которых дано ниже:

- `pwd`. Вывести имя текущего каталога.
- `ls`. Вывести содержимое каталога.
- `echo`. Вывести свой аргумент.
- `cat`. Вывести содержимое файла.
- `mkdir`. Создать каталог.
- `cp`. Скопировать файл.
- `rm`. Удалить файл.

Многие команды имеют ряд аргументов, это, в частности, касается `ls`, которая выше была вызвана с аргументом `-l`. Аргументы разделяются пробелами и имеют префикс `-`.

Узнать об аргументах, которые принимает команда, можно с помощью аргумента `--help`:

```

localhost:~# ls --help
BusyBox v1.31.1 () multi-call binary.

```

```

Usage: ls [-lAaCxdLHRFplinshrSXvctu] [-w WIDTH] [FILE]...

```

List directory contents

```

-1      One column output
-a      Include entries which start with .
-A      Like -a, but exclude . and ..
-x      List by lines
-d      List directory entries instead of contents
-L      Follow symlinks
-H      Follow symlinks on command line
-R      Recurse

```



```
-p      Append / to dir entries
-F      Append indicator (one of */=@|) to entries
-l      Long listing format
-i      List inode numbers
-n      List numeric UIDs and GIDs instead of names
-s      List allocated blocks
-lc     List ctime
-lu     List atime
--full-time    List full date and time
-h      Human readable sizes (1K 243M 2G)
--group-directories-first
-S      Sort by size
-X      Sort by extension
-v      Sort by version
-t      Sort by mtime
-tc     Sort by ctime
-tu     Sort by atime
-r      Reverse sort order
-w N    Format N columns wide
--color[={always,never,auto}]    Control coloring
```

Еще одним способом получить подробные сведения о конкретной команде является вызов вида `man <команда>`.

Без объяснений осталась строка `echo 'new file' > new_file.txt` в примере сеанса работы в командной строке выше. Здесь используется механизм перенаправления данных с помощью символов `<` (перенаправление ввода) и `>` (перенаправление вывода). В Linux имеется источник стандартного ввода `stdin` (код 0), а также два приемника стандартного вывода: `stdout` (код 1) и `stderr` (код 2, для ошибок). Организация ввода/вывода показана на рис. 6.

В примере ниже используется `stdout` и `stderr`:

```
localhost:~# pwd
/root
localhost:~# pwd > pwd.txt
localhost:~# pwd --foo
sh: pwd: illegal option --
localhost:~# pwd --foo 2> err.txt
localhost:~# cat err.txt
sh: pwd: illegal option --
```

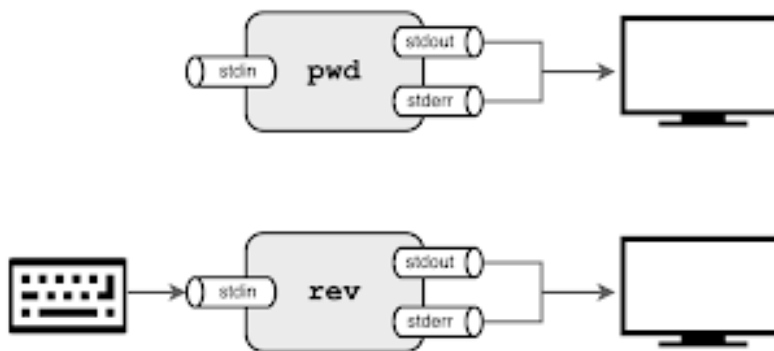


Рис. 6. Организация ввода/вывода

Обратите внимание на явное указание кода 2 при сохранении сообщения об ошибке в файл.

Перенаправление ввода/вывода превращается в очень мощную конструкцию при использовании такой организации команд, при которой вывод одной команды попадает на вход другой команды. Эта конструкция представляет собой конвейер и реализуется с помощью символа `|`, как показано в примере далее:

```
localhost:~# pwd > pwd.txt
localhost:~# rev --help
Usage: rev [options] [file ...]
```

Reverse lines characterwise.

Options:

```
-h, --help      display this help
-V, --version   display version
```

For more details see rev(1).

```
localhost:~# rev pwd.txt
toor/
localhost:~# pwd | rev
toor/
```

В Bash имеется удобный синтаксис для развертывания файловых путей (globbing). С помощью символов `*` (произвольная последовательность) и `?` (произвольный символ) реализуется подстановка имен файлов в духе регулярных выражений, как в примере ниже:

```
localhost:~# echo *
bench.py err.txt hello.c hello.js new_dir pwd.txt readme.txt rev
localhost:~# echo *.c
hello.c
localhost:~# echo p*
pwd.txt
localhost:~# echo *.*?
bench.py hello.js
```

В Bash есть возможность задать переменные и, кроме того, имеется ряд уже определенных переменных. Обратите внимание на особенности создания переменных:

```
localhost:~# A = 42
sh: A: not found
localhost:~# A=42
localhost:~# A
sh: A: not found
localhost:~# echo $A
42
```

С помощью команды `set` можно, помимо прочего, узнать, какие переменные сейчас заданы для текущего пользователя:

```
localhost:~# set
A='42'
HISTFILE='/root/.ash_history'
HOME='/root'
HOSTNAME='localhost'
IFS='
'
LINENO=''
OLDPWD='/'
OPTIND='1'
PAGER='less'
PATH='/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'
PPID='1'
PS1='\h:\w\$ '
PS2='> '
PS4='+ '
PWD='/root'
SHLVL='3'
```

```
TERM='linux'  
TZ='UTC-03:00'  
_='/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'  
script='/etc/profile.d/*.sh'
```

Особенно важна здесь переменная PATH, которая определяет те пути (разделенные с помощью :), где будет осуществляться поиск команд интерпретатором.

Linux является многопользовательской ОС и информация о зарегистрированных пользователях находится в конфигурационном файле /etc/passwd:

```
localhost:~# whoami  
root  
localhost:~# cat /etc/passwd  
root:x:0:0:root:/root:/bin/ash  
bin:x:1:1:bin:/bin:/sbin/nologin  
daemon:x:2:2:daemon:/sbin:/sbin/nologin  
adm:x:3:4:adm:/var/adm:/sbin/nologin  
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin  
sync:x:5:0:sync:/sbin:/bin/sync  
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown  
halt:x:7:0:halt:/sbin:/sbin/halt  
mail:x:8:12:mail:/var/mail:/sbin/nologin  
news:x:9:13:news:/usr/lib/news:/sbin/nologin  
uucp:x:10:14:uucp:/var/spool/uucppublic:/sbin/nologin  
operator:x:11:0:operator:/root:/sbin/nologin  
man:x:13:15:man:/usr/man:/sbin/nologin  
postmaster:x:14:12:postmaster:/var/mail:/sbin/nologin  
cron:x:16:16:cron:/var/spool/cron:/sbin/nologin  
ftp:x:21:21:ftp:/var/lib/ftp:/sbin/nologin  
sshd:x:22:22:sshd:/dev/null:/sbin/nologin  
at:x:25:25:at:/var/spool/cron/atjobs:/sbin/nologin  
squid:x:31:31:Squid:/var/cache/squid:/sbin/nologin  
xfs:x:33:33:X Font Server:/etc/X11/fs:/sbin/nologin  
games:x:35:35:games:/usr/games:/sbin/nologin  
cyrus:x:85:12:cy:/usr/cyrus:/sbin/nologin  
vpopmail:x:89:89:vm:/var/vpopmail:/sbin/nologin  
ntp:x:123:123:NTP:/var/empty:/sbin/nologin  
smmisp:x:209:209:smmisp:/var/spool/mqueue:/sbin/nologin
```

```

guest:x:405:100:guest:/dev/null:/sbin/nologin
nobody:x:65534:65534:nobody:/:/sbin/nologin
dhcp:x:100:101:dhcp:/var/lib/dhcp:/sbin/nologin
svn:x:101:102:svn:/var/svn:/sbin/nologin

```

Информация о каждом из пользователей занимает отдельную строку. Строка разделяется символом `:` на поля. Первое поле означает имя пользователя. В нашем случае это `root`. Последнее поле указывает путь к интерпретатору оболочки ОС. В нашем случае это компактный Bash-подобный интерпретатор `ash`.

Вспомним, как выглядит вывод команды `ls` в long-формате:

```

localhost:~# ls -l
total 20
-rw-r--r--    1 root    root          114 Jul  5  2020 bench.py
drwxr-xr-x    2 root    root           37 Nov  4 18:01 foo
-rw-r--r--    1 root    root           76 Jul  3  2020 hello.c
-rw-r--r--    1 root    root          22 Jun 26  2020 hello.js
-rw-r--r--    1 root    root          151 Jul  5  2020
↳  readme.txt

```

Первый столбец определяет права доступа и информацию о файле (`-`) или каталоге (`d`, как в случае с `foo`), закодированную в первом символе. Флаги доступа бывают следующих основных видов:

- `-`. Доступ запрещен.
- `r`. Имеется доступ на чтение.
- `w`. Имеется доступ на запись.
- `x`. Имеется доступ на исполнение (на вход в случае каталога).

Рассмотрим детали на примере с файлом `bench.py`, который имеет следующие права доступа:

```

-rw-r--r--    1 root    root          114 Jul  5  2020 bench.py
|[-][-][-]    [----]  [----]
| | | |      |      |
| | | |      |      +-----> 6. Группа
| | | |      +-----> 5. Владелец
| | | |
| | | +-----> 4. Права всех остальных
| | +-----> 3. Права группы
| +-----> 2. Права владельца

```

+-----> 1. Тип файла

При создании пользовательских команд необходимо указать права на исполнение, как показано в примере ниже:

```
localhost:~# echo "ls -l" > lsl
localhost:~# lsl
sh: lsl: not found
localhost:~# ./lsl
sh: ./lsl: Permission denied
localhost:~# chmod +x lsl
localhost:~# ./lsl
total 24
-rw-r--r--      1 root      root      114 Jul  5  2020 bench.py
drwxr-xr-x      2 root      root        37 Nov  4 18:01 foo
-rw-r--r--      1 root      root      76 Jul  3  2020 hello.c
-rw-r--r--      1 root      root     22 Jun 26  2020 hello.js
-rwxr-xr-x      1 root      root        6 Nov  4 18:44 lsl
-rw-r--r--      1 root      root     151 Jul  5  2020
↳  readme.txt
```

В Bash существует ряд специальных переменных, в частности:

- \$0. Путь к запущенной программе.
- \$1, \$2, .... Аргументы программы.
- \$#. Количество аргументов программы.
- \$@. Список аргументов программы.
- \$?. Значение результата выполнения программы (0 означает успешное выполнение).

Рассмотрим в качестве примера следующую программу tests.sh:

```
echo $0
echo $1 $2
echo $#
echo $@
```

Результат ее выполнения показан далее:

```
localhost:~# ./test.sh 1 2 3 4 5
./test.sh
1 2
5
1 2 3 4 5
```

```
localhost:~# echo $?  
0  
localhost:~# foo  
sh: foo: not found  
localhost:~# echo $?  
127
```

Рассмотрим теперь более сложный пример пользовательской команды. Далее приведен код на языке Bash вычисления факториала:

```
#!/bin/sh  
seq "$1" | xargs echo | tr " " "*" | bc
```

В первой строке указан интерпретатор, который будет использоваться для исполнения программы. По соглашению, такую строку необходимо всегда указывать первой в пользовательских скриптах. Далее используется ряд новых команд.

Команда `seq` генерирует последовательность чисел:

```
localhost:~# seq 5  
1  
2  
3  
4  
5
```

Команда `xargs` форматирует список из стандартного ввода:

```
localhost:~# seq 5 | xargs  
1 2 3 4 5
```

Команда `tr` осуществляет замену текстовых фрагментов:

```
localhost:~# seq 5 | xargs | tr " " "*"  
1*2*3*4*5
```

Команда `bc` представляет собой калькулятор:

```
localhost:~# echo "2+2" | bc  
4
```

Для вычислений в Bash можно также использовать скобки специального вида:

```
localhost:~# echo $((2 + 2))
```

4

Для получения результата команды в виде аргумента другой команды можно также использовать скобки специального вида:

```
localhost:~# echo "My folder is $(pwd)"
My folder is /root
```

В Bash имеются возможности полноценного языка программирования. Ниже приведен пример реализации факториала с использованием ветвлений и рекурсии:

```
#!/bin/sh
if [ "$1" -le 1 ] ; then
    echo 1
    return
fi
echo $(( $1 * $( ./fact.sh $(( $1 - 1 )) ) ))
```

Реализация факториала с использованием цикла:

```
#!/bin/sh
res=1
for i in $(seq 1 "$1"); do
    res=$((res * i))
done
echo $res
```

## 2.2. Инструменты grep, sed и awk

Команда `grep` осуществляет поиск по образцу, определяемому регулярным выражением. Команда `sed` является строчным редактором, но главное ее использование состоит в замене по шаблону, как и в случае `grep`, заданному регулярным выражением.

В табл. 1 показаны примеры некоторых базовых элементов регулярного выражения.

Таблица 1. Некоторые базовые регулярные выражения

Символ	Действие
Буквы, числа, некоторые знаки	Обозначают сами себя
.	Любой символ



Символ	Действие
[множество символов]	Любой символ из множества
[^множество символов]	Любой символ не из множества
^	Начало строки
\$	Конец строки
~	Начало строки
выражение*	Повторение выражения 0 или более раз
выражение выражение	Последовательность из выражений

С использованием `grep` и `sed` можно создать достаточно сложные схемы обработки данных. В частности, следующий код осуществляет проверку правописания для файла `README.md` на основе словаря из файла `unix-words`:

```
cat README.md | tr A-Z a-z | tr -cs A-Za-z '\n' | sort | uniq |
↪ grep -vx -f unix-words >out ; cat out | wc -l | sed 's/$/'
↪ misspelled words!/'
```

Еще более изощренным, чем `grep` и `sed`, является инструмент `awk`. `Awk` (по именам авторов – Aho, Weinberger, Kernighan) представляет собой язык программирования для обработки текстовых данных.

Ниже показан пример вывода колонки №5 из данных, предоставленных вызовом `ls -l`:

```
localhost:~# ls -l
total 36
-rw-r--r--      1 root      root           114 Jul  5  2020 bench.py
-rwxr-xr-x      1 root      root            51 Nov  4 18:56 fact.sh
-rwxr-xr-x      1 root      root            76 Nov  4 19:45 fact2.sh
drwxr-xr-x      2 root      root            37 Nov  4 18:01 foo
-rw-r--r--      1 root      root           76 Jul  3  2020 hello.c
-rw-r--r--      1 root      root           22 Jun 26  2020 hello.js
-rwxr-xr-x      1 root      root             6 Nov  4 18:44 lsl
-rw-r--r--      1 root      root          151 Jul  5  2020
↪  readme.txt
-rwxr-xr-x      1 root      root            36 Nov  4 18:50 test.sh
localhost:~# ls -l | awk '{ print $5 }'
```

76  
37  
76  
22  
6  
151  
36

Средствами awk легко подсчитать общий размер файлов:

```
localhost:~# ls -l | awk '{ s += $5 } END { print s }'
569
```

В заключение рассмотрим пример вывода на экран самой свежей новости с ресурса Hacker News:

```
#!/bin/sh
N=$(curl -s https://hacker-news.firebaseio.com/v0/topstories.json
↪ | jq '[0]')
curl -s "https://hacker-news.firebaseio.com/v0/item/$N.json" | jq
↪ '["title"]' | cowsay
```

Вот как выглядит вывод этой программы:

```
root@Server584432:~# ./hn.sh

-----
/ "Finishing my first game while working \
\ full-time"                               /
-----

      ^__^
      (oo)\_______
      (__)\       )\/\
           ||----w |
           ||     ||
```

# 3. Менеджеры пакетов

## 3.1. Нумерация версий ПО

**Программная библиотека** – контролируемая коллекция программных приложений и связанной с ними документации, предназначенная для использования в процессе разработки, эксплуатации и сопровождения программного обеспечения.

Библиотеки, модули, пакеты, расширения – элементы программных приложений, которые предназначены для модульной организации программы и повторного использования кода.

Существуют различные схемы нумерации версий:

- последовательные значения,
- по дате выпуска,
- различные экзотические схемы (TeX и проч.).

Указание стадии разработки в версии:

- начальная стадия разработки – альфа (например, 1.2.0-alpha.1),
- Стадия тестирования и отладки – бета (например, 1.2.0-beta),
- Выпуск-кандидат (например, 1.2.0-rc.3),
- Публичный выпуск (например, 1.2.0),
- Исправления после выпуска (1.2.5).

**Ад зависимостей** – проблема управления программными пакетами (библиотеками, модулями), которые зависят от определенных версий других программных пакетов.

В сложных случаях различные установленные программные продукты требуют наличия разных версий одной и той же библиотеки. В наиболее сложных случаях один продукт может косвенно требовать сразу две версии одной и той же библиотеки. Проблемы с зависимостями возникают у общих

пакетов/библиотек, у которых некоторые другие пакеты имеют зависимости от несовместимых и различных версий общих пакетов.

Без формальной спецификации номера версий практически бесполезны для управления зависимостями. Ясно определив и сформулировав идею версионирования, становится легче сообщать о намерениях пользователям ПО.

В настоящее время популярность приобрела схема семантической нумерации версий рис. 7.



Рис. 7. Семантическая нумерация версий

Учитывая номер версии МАЖОРНАЯ.МИНОРНАЯ.ПАТЧ, следует увеличивать:

1. МАЖОРНУЮ версию, когда сделаны обратно несовместимые изменения API.
2. МИНОРНУЮ версию, когда вы добавляете новую функциональность, не нарушая обратной совместимости.
3. ПАТЧ-версию, когда вы делаете обратно совместимые исправления.

Дополнительные обозначения для предрелизных и билд-метаданных возможны как дополнения к МАЖОРНАЯ.МИНОРНАЯ.ПАТЧ формату.

Обычный номер версии ДОЛЖЕН иметь формат X.Y.Z, где X, Y и Z – неотрицательные целые числа и НЕ ДОЛЖНЫ начинаться с нуля. X – мажорная версия, Y – минорная версия и Z – патч-версия. Каждый элемент ДОЛЖЕН увеличиваться численно. Например: 1.9.0 -> 1.10.0 -> 1.11.0.

После релиза новой версии пакета содержание этой версии НЕ ДОЛЖНО быть модифицировано. Любые изменения ДОЛЖНЫ быть выпущены как новая версия.

**Мажорная версия X** ( $X.y.z \mid X > 0$ ) ДОЛЖНА быть увеличена, если в публичном API представлены какие-либо обратно несовместимые изменения. Она МОЖЕТ включать в себя изменения, характерные для уровня

минорных версий и патчей. Когда увеличивается мажорная версия, минорная и патч-версия ДОЛЖНЫ быть обнулены.

Мажорная версия ноль (0.y.z) предназначена для начальной разработки. Всё может измениться в любой момент. Публичный API не должен рассматриваться как стабильный.

**Минорная версия** (x.y.z | x > 0) ДОЛЖНА быть увеличена, если в публичном API представлена новая обратно совместимая функциональность. Версия ДОЛЖНА быть увеличена, если какая-либо функциональность публичного API помечена как устаревшая (deprecated). Версия МОЖЕТ быть увеличена в случае реализации новой функциональности или существенного усовершенствования в приватном коде. Версия МОЖЕТ включать в себя изменения, характерные для патчей. Патч-версия ДОЛЖНА быть обнулена, когда увеличивается минорная версия.

**Патч-версия Z** (x.y.Z | x > 0) ДОЛЖНА быть увеличена только если содержит обратно совместимые баг-фиксы. Определение баг-фикс означает внутренние изменения, которые исправляют некорректное поведение.

Приоритет определяет, как версии соотносятся друг с другом, когда упорядочиваются. Приоритет версий ДОЛЖЕН рассчитываться путем разделения номеров версий на мажорную, минорную, патч и предрелизные идентификаторы. Приоритет определяется по первому отличию при сравнении каждого из этих идентификаторов слева направо: Мажорная, минорная и патч-версия всегда сравниваются численно. Пример: 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1. Когда мажорная, минорная и патч-версия равны, предрелизная версия имеет более низкий приоритет, чем нормальная версия.

## 3.2. Управление пакетами

Менеджер пакетов – ПО для управления процессом установки, удаления, настройки и обновления различных компонентов программного обеспечения. Менеджеры пакетов бывают следующих видов:

- уровня ОС (например, RPM в Linux),
- уровня языка программирования (например, npm для JavaScript),
- уровня приложения (например, Package Control для Sublime Text, управление плагинами для Eclipse).

Пакет содержит в некотором сжатом файловом формате программный код и метаданные. К метаданным относятся:

- указание авторства, версии и описание пакета,
- список зависимостей пакета,
- хэш-значение содержимого пакета.
- Пример метаданных из пакета для npm (JavaScript):

```
{
  "name": "my_package",
  "version": "1.0.0",
  "dependencies": {
    "my_dep": "^1.0.0",
    "another_dep": "~2.2.0"
  }
}
```

Пример метаданных из пакета для NuGet (.NET):

```
<?xml version="1.0" encoding="utf-8"?>
<package
  xmlns="http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd">
  <metadata>
    <id>sample</id>
    <version>1.0.0</version>
    <authors>Microsoft</authors>
    <dependencies>
      <dependency id="another-package" version="3.0.0" />
      <dependency id="yet-another-package" version="1.0.0"
    />
    </dependencies>
  </metadata>
</package>
```

Пример метаданных из пакета для Maven (Java):

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  https://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
```

```
<artifactId>junit</artifactId>
<version>4.12</version>
<type>jar</type>
<scope>test</scope>
<optional>true</optional>
</dependency>
...
</dependencies>
...
</project>
```

Пример проблемы с зависимостями пакетов – ромбовидная зависимость, см. рис. 8.

Пакет А нуждается в пакетах В и С. В нужна версия 1 пакета D, а не версия 2. С требует D версии 2, а не версии 1. В этом случае, если предположить, что невозможно выбрать обе версии D, нет возможности построить А.

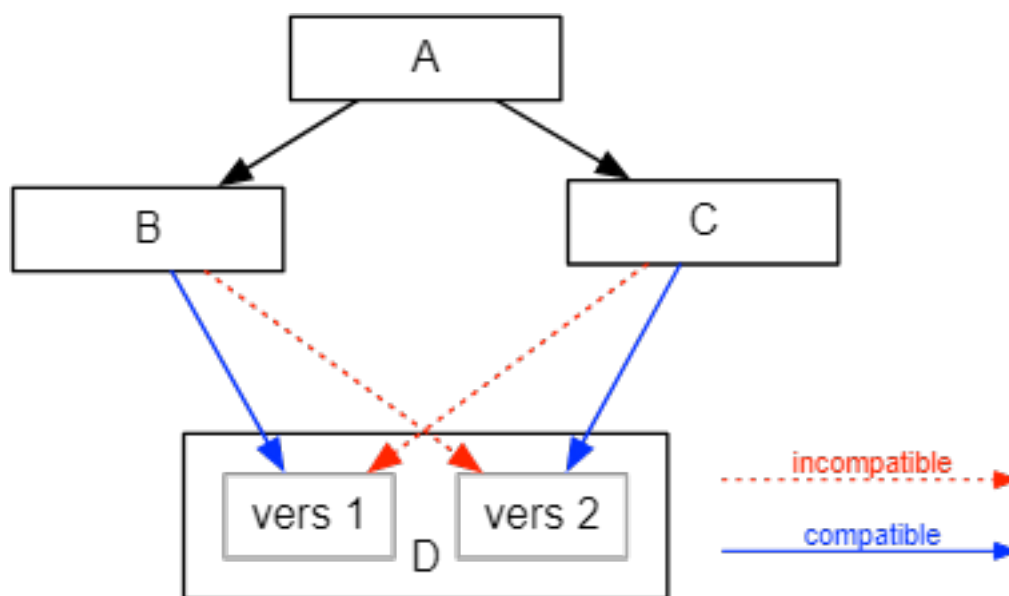


Рис. 8. Пример ромбовидной зависимости между пакетами

В общем случае задача разрешения зависимостей между пакетами является NP-полной. Для ее решения сегодня применяются SAT-решатели.

# 4. Конфигурационные языки

## 4.1. Виды конфигурационных языков

Параметры конфигурации ПО могут сохраняться в специальном виде, доступном для редактирования пользователями и другими программами. Для хранения может использоваться специальная база данных, но наиболее распространенным вариантом хранения настроек программы являются файлы конфигурации, содержимое которых написано на одном из конфигурационных языков.

В качестве конфигурационных языков могут использоваться:

- непрограммируемые форматы описания конфигурации,
- языки программирования общего назначения,
- программируемые DSL для задач конфигурирования.

Формальный язык – множество конечных слов над конечным алфавитом.

Языки программирования определяются с помощью синтаксиса и семантики:

- синтаксис – форма или структура выражений, предложений и программных единиц,
- семантика – значение выражений, предложений и программных единиц.



## 4.2. Представление синтаксиса языка с помощью БНФ

Форма Бэкуса-Наура (БНФ) – формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории. В БНФ присутствуют:

- нетерминальные символы: абстракции BNF.
- терминальные символы: лексемы.
- грамматика: набор правил.

У правила есть левая и правая стороны; правило состоит из терминальных и нетерминальных символов. Пример БНФ-описания арифметического выражения:

```
<expr> ::= <term> "+" <expr>
          | <term>
```

```
<term> ::= <factor> "*" <term>
          | <factor>
```

```
<factor> ::= "(" <expr> ")"
           | <const>
```

```
<const> ::= integer
```

## 4.3. Простые форматы описания конфигурации

Одним из старейших форматов являются *S-выражения* языка Lisp. Это нотация для древовидных структур, реализованных в виде вложенных списков. В Lisp S-выражения применяются и для данных, и для программ. S-выражение, заключенное в скобки, определяется индуктивно, как одна из альтернатив:

- атом,
- выражение вида (x . y), где x и y – S-выражения.

В БНФ:

```
<s-exp> ::= <atom>
```

```

    | '(' <s-exp-list> ')'

<s-exp-list> ::= <sexp> <s-exp-list>
    |

<atom> ::= <symbol>
    | <integer>
    | #t | #f

```

Пример S-выражения:

```

(users
  ((uid 1) (name root) (gid 1))
  ((uid 108) (name peter) (gid 108))
  ((uid 109) (name alex) (gid 109)))

```

К недостаткам S-выражений для описания конфигурации относятся:

- сложность чтения для человека синтаксиса с большим числом скобок,
- нет стандарта S-выражений для описания конфигурационных данных.

Классическим способом представления файлов конфигурации является формат INI из Windows, а также схожие с ним варианты conf из Unix. В основе формата – пара ключ-значение. Пары схожего назначения объединяются в секции:

```

[секция_1]
параметр1=значение1
параметр2=значение2

```

```

[секция_2]
параметр1=значение1
параметр2=значение2

```

В (расширенной) БНФ:

```

ini ::= {section}
section ::= "[" name "]" "\n" {entry}
entry ::= key "=" value "\n"

```

Здесь фигурные скобки обозначают повторение 0 или более раз.

Пример INI-файла (system.ini):

```

; for 16-bit app support

```

```
[386Enh]
woafont=dosapp.fon
EGA80WOA.FON=EGA80WOA.FON
EGA40WOA.FON=EGA40WOA.FON
CGA80WOA.FON=CGA80WOA.FON
CGA40WOA.FON=CGA40WOA.FON
```

```
[drivers]
wave=mmdrv.dll
timer=timer.driv
```

```
[mci]
[network]
Bios=1438818414
SSID=29519693
```

К недостаткам INI для описания конфигурации относятся: \* отсутствие поддержки вложенных конструкций, \* нет стандарта INI для описания конфигурационных данных.

Еще недавно наибольшую популярность среди форматов конфигурационных данных имел **XML** (eXtensible Markup Language).

XML разрабатывался как язык с простым формальным синтаксисом, удобный для создания и обработки документов программами и одновременно удобный для чтения и создания документов человеком, с подчеркиванием нацеленности на использование в Интернете.

В самом базовом виде XML напоминает S-выражения, в которых вместо скобок используются открывающийся и закрывающийся именованные теги.

Грамматика XML в БНФ здесь не приводится из-за ее большого объема.

Пример XML-данных:

```
<?xml version="1.0" encoding="UTF-8"?>
<shiporder orderid="889923"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="shiporder.xsd">
  <orderperson>John Smith</orderperson>
  <shipto>
    <name>Ola Nordmann</name>
    <address>Langgt 23</address>
```

```

    <city>4000 Stavanger</city>
    <country>Norway</country>
</shipto>
<item>
    <title>Hide your heart</title>
    <quantity>1</quantity>
    <price>9.90</price>
</item>
</shiporder>

```

Важной особенностью XML является поддержка специальных схем для определения корректности XML-данных: DTD (Document Type Definition), XML Schema и другие. DTD определяет:

- состав элементов, которые могут использоваться в XML документе;
- описание моделей содержания, т.е. правил вхождения одних элементов в другие;
- состав атрибутов, с какими элементами XML документа они могут использоваться;
- каким образом атрибуты могут применяться в элементах;
- описание сущностей, включаемых в XML документ.

В XML Schema дополнительно введены типы данных для элементов. Фрагмент XML Schema описания элемента item из примера выше:

```

<xs:element name="item" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="note" type="xs:string" minOccurs="0"/>
      <xs:element name="quantity" type="xs:positiveInteger"/>
      <xs:element name="price" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

К недостаткам XML для описания конфигурации относится:

- высокая сложность формата, как для чтения человеком, так и для программного разбора.

Формат **JSON** (JavaScript Object Notation) основан на синтаксисе JavaScript. Изначально он использовался в качестве текстового формата

обмена данными, но со временем все чаще стал применяться и в качестве формата описания конфигурации приложения.

В JSON используются следующие типы данных:

- число в формате double,
- строка,
- логическое значение true или false,
- массив значений любого поддерживаемого типа,
- объект – словарь пар ключ-значение, где ключ – строка, а значение – любой поддерживаемый тип.
- специальное значение null.

Пример JSON-данных:

```
{  
  "menu": {  
    "id": "file",  
    "value": "File",  
    "popup": {  
      "menuitem": [  
        {"value": "New", "onclick": "CreateNewDoc()"},  
        {"value": "Open", "onclick": "OpenDoc()"},  
        {"value": "Close", "onclick": "CloseDoc()"}  
      ]  
    }  
  }  
}
```

Для проверки корректности JSON-данных в формат позже была добавлена схема – JSON Schema, которая во многом схожа с XML Schema. К недостаткам JSON для описания конфигурации относится:

- отсутствие поддержки комментариев.

Формат **YAML** (YAML Ain't Markup Language) предназначен для сериализации данных, но также часто используется в качестве конфигурационного языка. YAML с точки зрения синтаксиса имеет сходство с языком Python – для определения вложенности конструкций используются отступы. Особенностью YAML является возможность создания (с помощью &) и использования (с помощью \*) ссылок на элементы документа. Это позволяет не приводить полностью повторно встречающиеся данные.

Пример YAML-файла:

```

receipt:      Oz-Ware Purchase Invoice
date:         2012-08-06
customer:
  first_name:  Dorothy
  family_name: Gale

items:
  - part_no:   A4786
    descrip:   Water Bucket (Filled)
    price:     1.47
    quantity:  4

  - part_no:   E1628
    descrip:   High Heeled "Ruby" Slippers
    size:      8
    price:     133.7
    quantity:  1

bill-to:  &id001
  street: |
    123 Tornado Alley
    Suite 16
  city:   East Centerville
  state:  KS

ship-to:  *id001
...

```

К недостаткам YAML для описания конфигурации относятся:

- сложность редактирования сильно вложенных элементов, проблемы с отступом,
- сложность стандарта YAML (размер файла спецификации больше, чем у XML).

Формат **TOML** (Tom's Obvious, Minimal Language) специально предназначен для использования в конфигурационных файлах. Его синтаксис основан на INI. Спецификация TOML не имеет БНФ-описания.

Пример TOML-данных:

```
# This is a TOML document
```

```
title = "TOML Example"

[owner]
name = "Tom Preston-Werner"
dob = 1979-05-27T07:32:00-08:00

[database]
enabled = true
ports = [ 8001, 8001, 8002 ]
data = [ ["delta", "phi"], [3.14] ]
temp_targets = { cpu = 79.5, case = 72.0 }

[servers]

  [servers.alpha]
  ip = "10.0.0.1"
  role = "frontend"

  [servers.beta]
  ip = "10.0.0.2"
  role = "backend"
```

К недостаткам TOML для описания конфигурации относится:

- сложность спецификации языка.

## 4.4. Языки общего назначения

Использование языка программирования общего назначения для описания конфигурации ПО обладает рядом преимуществ по сравнению с простыми непрограммируемыми конфигурационными форматами:

- поддержка функций,
- поддержка циклов,
- поддержка типов данных,
- поддержка импорта файлов.

Благодаря поддержке функций, циклов и разбиению файлов на модули достигается принцип DRY (Don't repeat yourself) – не допускать повторения одной и той же информации в программе. В результате конфигурационные

файлы имеют компактную, удобную для редактирования форму.

Поддержка типов данных, в том числе пользовательских, позволяет осуществлять на уровне типов проверку корректности программы – то, для чего в таких форматах, как XML, используются схемы.

Некоторые примеры использования языков общего назначения в качестве конфигурационных языков:

- В текстовом редакторе Emacs используется встроенный язык Lisp для описания конфигурации.
- В БД Tarantool используется Lua.
- Конфигурации на языке Python используются в `setuptools` и `Jupyter`.
- В Gradle конфигурация сборки может быть описана на Groovy и Kotlin.

К недостаткам языков общего назначения для задач конфигурирования относятся: \* не безопасное исполнение стороннего кода, \* сложность, чрезмерная выразительность языка (Тьюринг-полнота), мешающая задачам анализа и порождения конфигурационных данных.

## 4.5. Программируемые DSL для задач конфигурирования

Одним из наиболее интересных программируемых конфигурационных языков является Dhall. Этот язык со статической типизацией, основанный на принципах функционального программирования. Авторы определяют Dhall, как JSON + функции + типы + импорты. Важной особенностью Dhall является тот факт, что это не Тьюринг-полный язык. Хотя в нем и есть циклы, их бесконечное исполнение не поддерживается. В Dhall имеется режим нормализация представления, при котором все языковые абстракции раскрываются и результат представляет собой уровень представления JSON или YAML.

Пример описания данных на Dhall:

```
let makeUser = \(user : Text) ->
  let home      = "/home/${user}"
  let privateKey = "${home}/.ssh/id_ed25519"
  let publicKey  = "${privateKey}.pub"
  in { home = home
      , privateKey = privateKey
      , publicKey  = publicKey
```



```

    }
    {- Add another user to this list -}
in [ makeUser "bill"
    , makeUser "jane"
  ]

```

Результат преобразования в JSON:

```

[
  {
    "home": "/home/bill",
    "privateKey": "/home/bill/.ssh/id_ed25519",
    "publicKey": "/home/bill/.ssh/id_ed25519.pub"
  },
  {
    "home": "/home/jane",
    "privateKey": "/home/jane/.ssh/id_ed25519",
    "publicKey": "/home/jane/.ssh/id_ed25519.pub"
  }
]

```

Другими примерами программируемых конфигурационных языков являются: Nix, CUE (типы с ограничениями) и Jsonnet.

## 5. Системы автоматизации сборки

**Автоматизация сборки** – автоматизация создания программного обеспечения и связанные с этим процессы, включая компиляцию исходного кода, упаковку кода в дистрибутив и выполнение автоматических тестов.

**Системы сборки** автоматизируют выполнение повторяемых задач как на уровне отдельных пользователей, так и для крупных организаций.

Система сборки является **минимальной**, если она выполняет каждую задачу не более одного раза в процессе сборки, при этом задачи выполняются только в том случае, если они прямо или косвенно зависят от входных данных, которые изменились с момента предыдущей сборки.

Важнейшим элементом системы сборки является **граф зависимостей задач**, см. рис. 9.

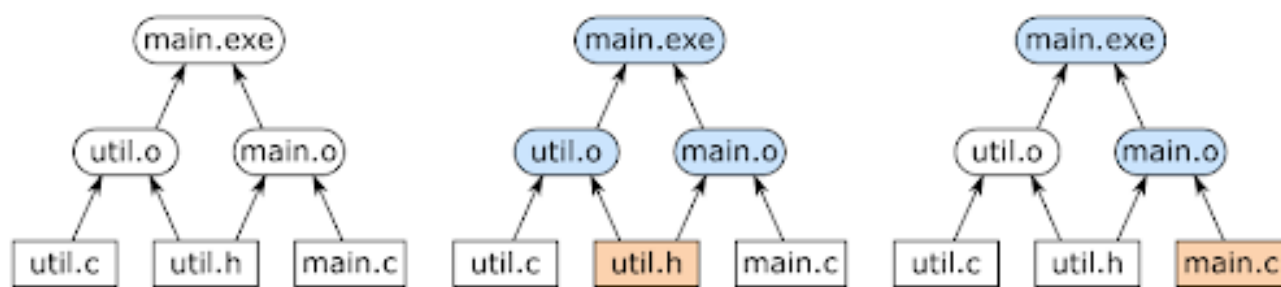


Рис. 9. Граф зависимостей задач и два варианта сборки: а) полное перестроение, б) частичное перестроение.

Система сборки включает в себя следующие элементы:

- Хранилище пар ключ-значение. Цель системы сборки привести хранилище в актуальное состояние. Во многих случаях хранилищем является файловая система, где ключами являются имена файлов, а зна-

чениями – содержимое файлов. Многие системы сборки используют хеш-суммы от значений для компактного описания данных и быстрой проверки данных на равенство.

- Входные, выходные и промежуточные значения. Входные значения могут предоставляться пользователем или зависеть от выполнения других задач. Выходные значения формируются в результате выполнения задачи. Промежуточные значения формируются в процессе выполнения задачи.
- Постоянная информация о сборке. Эта информация сохраняется от между сборками и представляет собой “память” системы сборки.
- Описание задачи. Пользователь предоставляет описание того, каким образом выходное значение по заданному ключу получается на основе входных зависимостей.

Система сборки принимает описание задачи, целевой ключ и хранилище и возвращает измененное хранилище, в котором целевой ключ и всего его зависимости принимают актуальные значения.

Постоянная информация о сборке может храниться в одном из следующих вариантов:

- Время модификации файлов в файловой системе. Если время модификации одного из файлов-зависимостей задачи более новое, чем время модификации файла-результата самой задачи – необходимо перестроить задачу.
- Специальный флаг для каждого ключа хранилища, сигнализирующий о том, что данные были обновлены. После сборки все флаги сбрасываются. Когда начинается новая сборка для всех ключей, значения которых изменились после прошлой сборки, устанавливается флаг. Если для ключа и всех его прямых и косвенных зависимостей флаг сброшен, то ключ перестраивать не требуется.
- Граф зависимостей, в котором хранятся хеш-значения файлов-зависимостей.
- Кэш-хранилище с адресацией по хеш-значению файла, а также история всех предыдущих сборок с указанием хеш-значений файлов для каждой задачи. Этот вариант используется для облачного (распределенной) режима работы с системой сборки силами коллектива разработчиков.

Системы сборки различаются по типу используемого **алгоритма планировщика**:

- Топологическая сортировка,
- Выполнение с рестартами задач.
- Выполнение с приостановкой задач.

Также системы сборки различаются по типу зависимостей:

- Статические зависимости.
- Динамические зависимости.

**Динамические зависимости** образуются в процессе вычислений и не могут быть определены заранее, до начала выполнения сборки.

На рис. 10 показан пример динамической зависимости. В файле `release.txt` находятся все файлы выпуска. Этот файл образуется на основе слияния файлов `bins.txt` и `docs.txt` (документация). Файл `release.tar` является архивом выпуска. Зависимости `release.tar` не заданы статически. Они определяются содержимым файла `release.txt`. По этой причине, если информация о файле `README` добавляется в `docs.txt`, то необходима поддержка динамических зависимостей для правильной сборки `release.tar`.

Кроме того, в системе сборки может применяться **техника раннего среза** (*early cutoff*) – если задача выполнена, но ее результат не изменился с предыдущей сборки, то нет необходимости исполнять зависимые задачи, то есть процесс сборки можно завершить ранее.

На рис. 11 показан пример раннего среза. Если в `main.c` был добавлен новый комментарий, тогда сборка может быть остановлена после определения отсутствия изменений в `main.o`.

При использовании облачной системы сборки скорость сборки может быть существенно увеличена, благодаря разделению результатов сборки между участниками команды. При этом облачная сборка может поддерживать вариант сборки, при котором локально образуются только конечные результаты сборки, а все промежуточные файлы остаются в облаке.

На рис. 12 приведен пример сценария работы с облачной системой сборки. Пользователь совершает следующие действия:

- Загружает исходные тексты, их хеш-значения 1, 2 и 3. Затем пользователь запрашивает сборку `main.exe`. Система сборки определяет с помощью изучения истории предыдущих сборок, что кто-то уже скомпилировал ранее именно эти исходные тексты. Результаты их сборки хранятся в облаке с хешами 4 (`util.o`) и 5 (`main.o`). Система сборки далее определяет, что для зависимостями с такими хешами есть `main.exe`

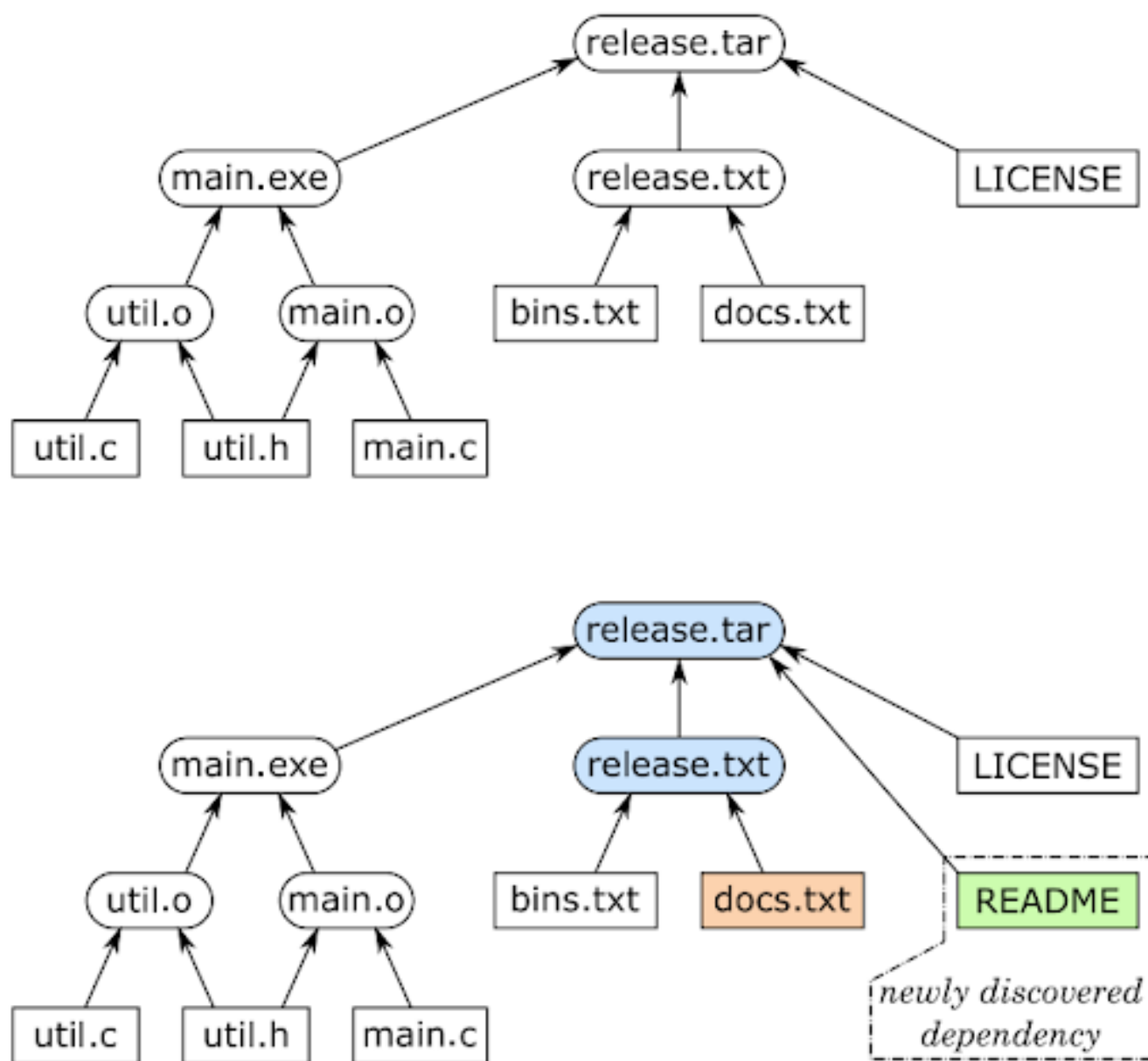


Рис. 10. Пример использования динамической зависимости

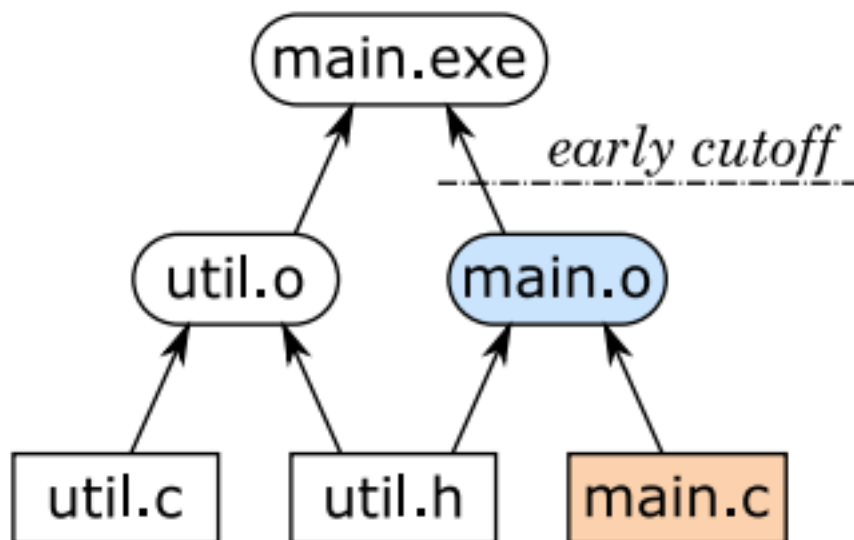


Рис. 11. Пример оптимизации раннего среза

с хешем 6. По ключу 6 из облачного хранилища извлекается конечный результат.

- Далее пользователь изменяет util.c, и его хеш становится равен 7. В облаке комбинации хешей (7, 2) не существует, то есть ранее никто еще не компилировал такой вариант исходного кода. Процесс продолжается до получения нового main.exe, после чего новые варианты файлов и их хеш-значения сохраняются в облаке.

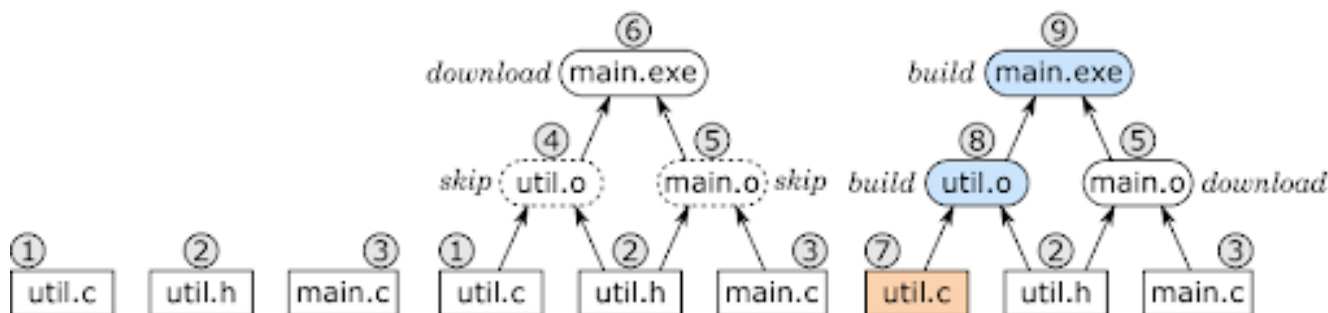


Рис. 12. Пример сценария работы с облачной системой сборки

## 5.1. Топологическая сортировка графа зависимостей

**Топологической сортировкой** называют порядок нумерации вершин ориентированного графа, при котором любое ребро идет из вершины с мень-

шим номером в вершину с большим. Очевидно, что не любой граф можно отсортировать топологически. Можно доказать, что топологическая сортировка существует для ациклических графов и не существует для циклических.

Пример топологической сортировки показан на рис. 13.

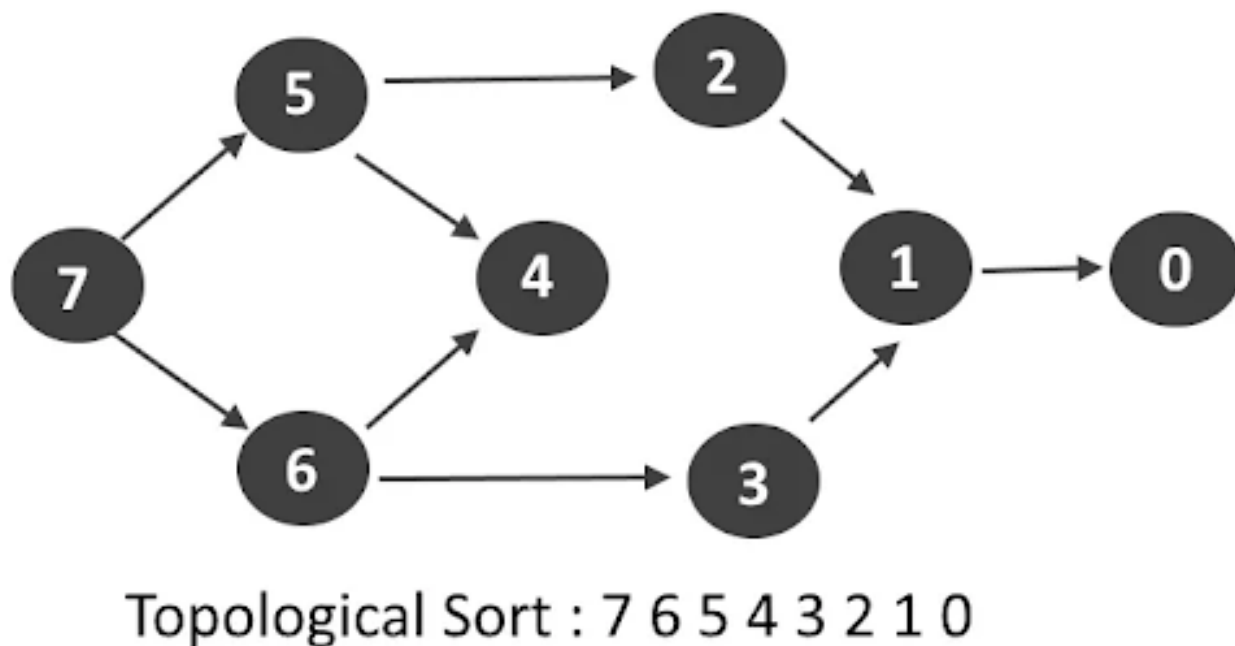


Рис. 13. Граф и результат его топологической сортировки

Алгоритм топологической сортировки на основе обхода графа в глубину:

```
// G - исходный граф
function topologicalSort():
    // проверить граф G на ацикличность
    fill(visited,false)
    for v in V(G)
        if not visited[v]
            dfs(v)
    ans.reverse()

function dfs(u):
    visited[u]=true
    for (u,v) in E(G)
        if not visited[v]
            dfs(v)
```

```
ans.pushBack(u)
```

## 5.2. Система сборки make

Утилита make считывает из специального файла с именем Makefile или makefile в текущей директории инструкции о том, как (при помощи каких команд) компилировать и собирать программы, а также информацию, из каких файлов состоит программа, которую надо «сделать».

Одним из главных достоинств make (чрезвычайно полезным при создании больших программ) является то, что он сравнивает времена модификации файлов, и если, к примеру, файл file1.c новее, чем получаемый из него file1.o, то make поймет, что перекомпилировать надо только его, а остальные – не нужно (если они не изменились).

Makefile содержат три основных компонента:

- Правила, как из одних файлов создавать другие (например, .o из .c).
- Так называемые «зависимости», которые указывают, что, например, исполняемый файл proggie собирается из файлов prg\_main.o и prg\_funcs.o, а те, в свою очередь, получаются из файлов prg\_main.c и prg\_funcs.c.
- Определения переменных, позволяющие делать Makefile более гибкими.

Ниже приведен пример простейшего Makefile:

```
CC = gcc
CFLAGS = -c -W -Wall

%.o : %.c
    $(CC) $(CFLAGS) -o $@ $<

all:
    program

program:
    prg_main.o prg_funcs.o
    $(CC) -o proggie prg_main.o prg_funcs.o
```

**Определения переменных.** Строки вида ИМЯ=значение – это определения переменных. Для получения значения переменной используется запись \$(ИМЯ) (знак доллара, а за ним имя переменной в скобках). Переменная может определяться через значения других переменных, например:



```
CFLAGS= -c $(WARNINGOPTIONS)
```

**Правила.** Запись `%.o : %.c` с последующей командой означает: «для любого файла `.c`, чтобы из него получить одноименный файл `.o`, надо выполнить такую-то команду», в данном случае:

```
gcc -c -W -Wall -o $@ $<
```

В правилах всегда используются специальные переменные `$@` и `$<`. Переменная `$@` обозначает «тот файл, который надо получить» (в данном случае `.o`), а `$<` – «исходный файл» (в данном случае `.c`). Такие переменные называются автоматическими. В примере:

```
all: library.cpp main.cpp
```

- `$@` обозначает `all`,
- `$<` обозначает `library.cpp`,
- `$$` обозначает `library.cpp main.cpp`.

Команда располагается на следующей строке, причем эта строка обязательно должна начинаться с символа табуляции.

Это довольно странное ограничение является основным источником ошибок и запутанности Makefile'ов – ведь визуально отличить символ табуляции от цепочки пробелов невозможно.

**Зависимости.** Запись вида

```
program: prg_main.o prg_funcs.o
```

означает, что файл `program` зависит от файлов `prg_main.o` и `prg_funcs.o`. Файл `program` называется целью (`target`), а файлы `.o` – зависимостями (`dependencies`). В системе `make` используется планировщик на основе алгоритма топологической сортировки, а постоянная информация о сборке извлекается из времени модификации файлов.

Эволюция систем автоматизации сборки показана на рис. 14.

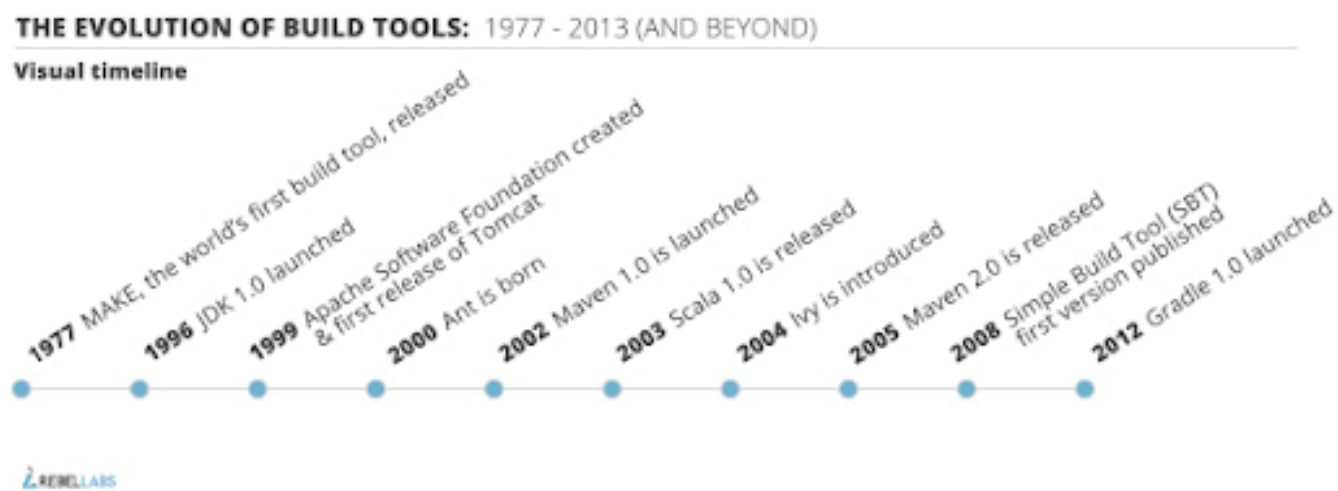


Рис. 14. Эволюция систем сборки

## 6. Системы контроля версий

**Системы контроля версий** – это класс программных систем, отвечающих за управление изменениями в компьютерных программах, документах, веб-сайтах и других наборах данных. Система контроля версий отслеживает изменения в файл или набор файлов в течение времени и позволяет вернуться позже к определённой версии файлов.

**Коммит** – фиксация факта изменений в системе контроля версий (СКВ).

В системе контроля версий обычно выделяются следующие функции:

- Хранение данных.
- Отслеживание изменений в данных (история, включая метаданные слияния).
- Распространение данных и истории изменений среди соавторов проекта.

Два наиболее распространенных варианта учета изменений данных в СКВ:

- набор изменений на основе различий (дельты),
- представление данных в виде направленного ациклического графа.

Системы CVS, Subversion, Perforce, Bazaar и т.д. представляют хранимую информацию в виде набора файлов и изменений, сделанных в каждом файле, по времени (обычно это называют контролем версий, основанным на различиях), см. рис. 15.

Другой подход к хранению данных больше похож на набор снимков миниатюрной файловой системы. Каждый раз, когда вы делаете коммит, то есть сохраняете состояние своего проекта, СКВ запоминает, как выглядит

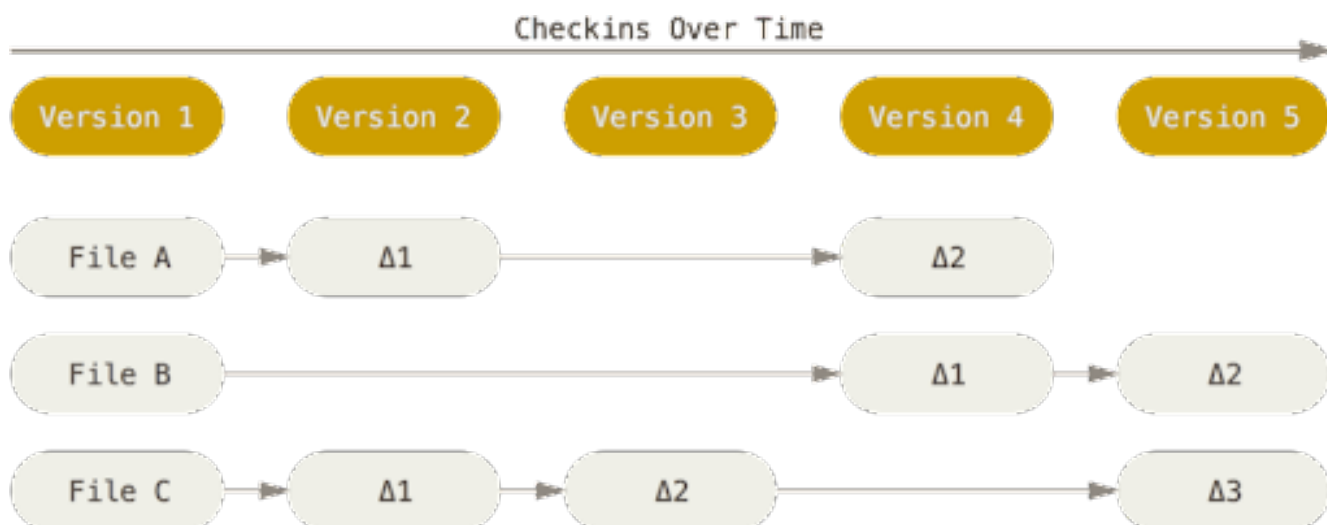


Рис. 15. Хранение данных как набора изменений относительно первоначальной версии каждого из файлов

каждый файл в этот момент, и сохраняет ссылку на этот снимок. Для увеличения эффективности, если файлы не были изменены, СКВ не запоминает эти файлы вновь, а только создает ссылку на предыдущую версию идентичного файла, который уже сохранен. СКВ представляет свои данные как граф снимков, см. рис. 16.

С точки зрения типа используемой истории изменений системы контроля версий различаются:

- использованием линейной истории,
- использованием графового представления для истории изменений.

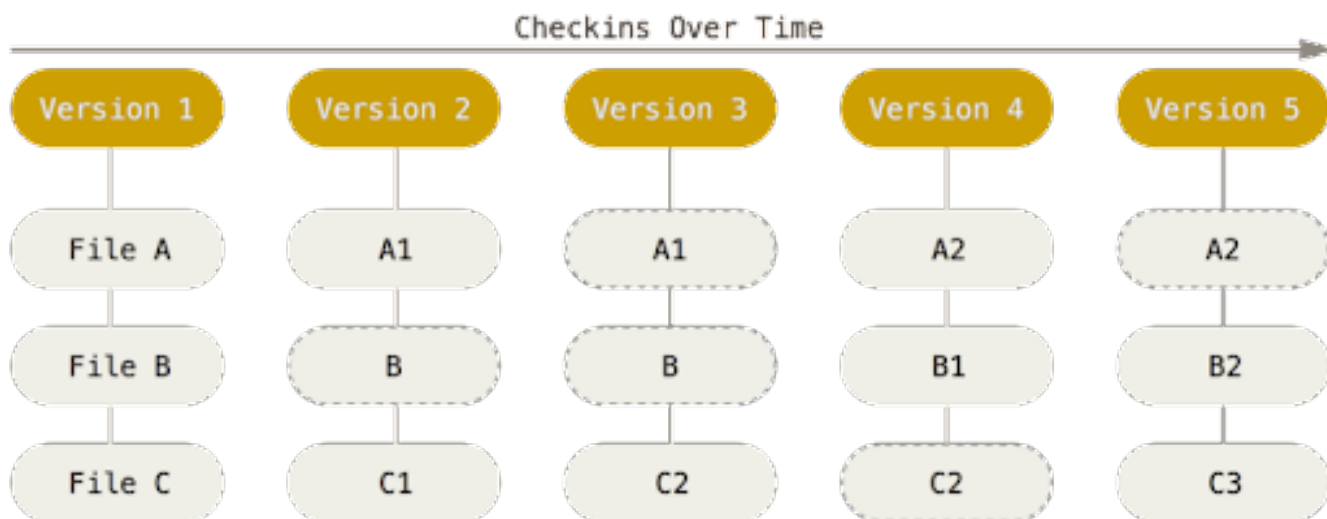


Рис. 16. Хранение данных как снимков проекта во времени

Системы контроля версий бывают различаются с точки зрения хранения

и доставки своего содержимого:

- Локальные системы.
- Централизованные системы.
- Распределенные системы.

**Локальные системы контроля версий.** Многие люди в качестве метода контроля версий применяют копирование файлов в отдельную директорию (возможно даже, директорию с отметкой по времени). Данный подход очень распространён из-за его простоты, однако он невероятно сильно подвержен появлению ошибок. Можно легко забыть, в какой директории вы находитесь, и случайно изменить не тот файл или скопировать не те файлы, которые вы хотели. Для того, чтобы решить эту проблему, были разработаны локальные СКВ с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий, см. рис. 17.

Одной из популярных СКВ была система RCS, которая и сегодня распространяется со многими компьютерами. RCS хранит на диске наборы патчей (различий между файлами) в специальном формате, применяя которые она может воссоздавать состояние каждого файла в заданный момент времени.

Локальные СКВ страдают от следующей проблемы: когда вся история проекта хранится в одном месте, есть риск потерять все.

**Централизованные системы контроля версий.** Следующая серьезная проблема, с которой сталкиваются разработчики, – это необходимость взаимодействовать с другими разработчиками. Для того, чтобы разобраться с ней, были разработаны централизованные системы контроля версий (ЦСКВ). Такие системы, как CVS, Subversion и Perforce, используют единый сервер, содержащий все версии файлов, и некоторое количество клиентов получают файлы из этого централизованного хранилища, см. рис. 18. Применение ЦСКВ являлось стандартом на протяжении многих лет.

Такой подход имеет множество преимуществ, особенно перед локальными СКВ. Например, все разработчики проекта в определенной степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать ЦСКВ, чем оперировать локальными базами данных на каждом клиенте.

Несмотря на это, данный подход тоже имеет серьезные минусы. Самый очевидный минус – это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого

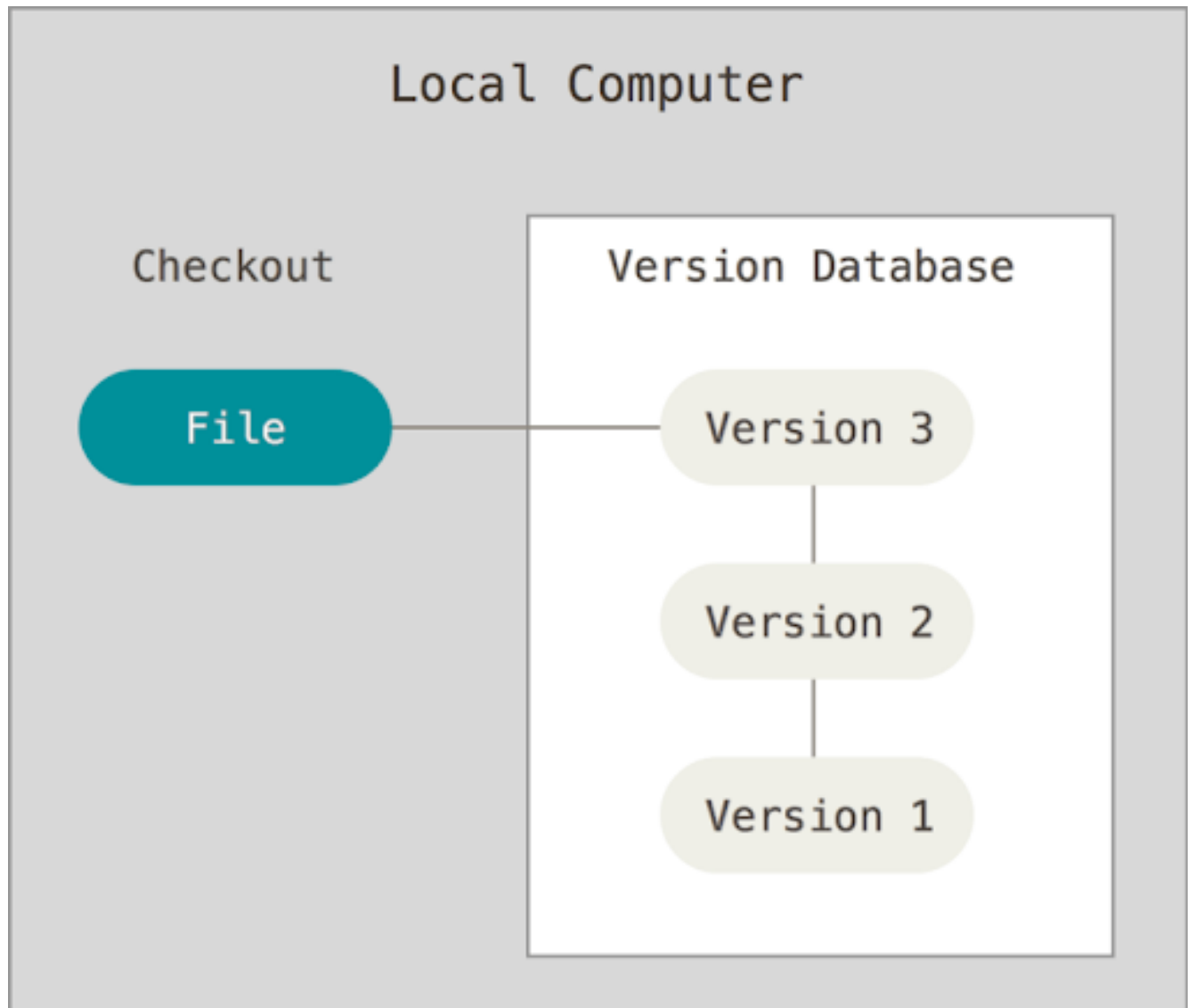


Рис. 17. Локальный контроль версий

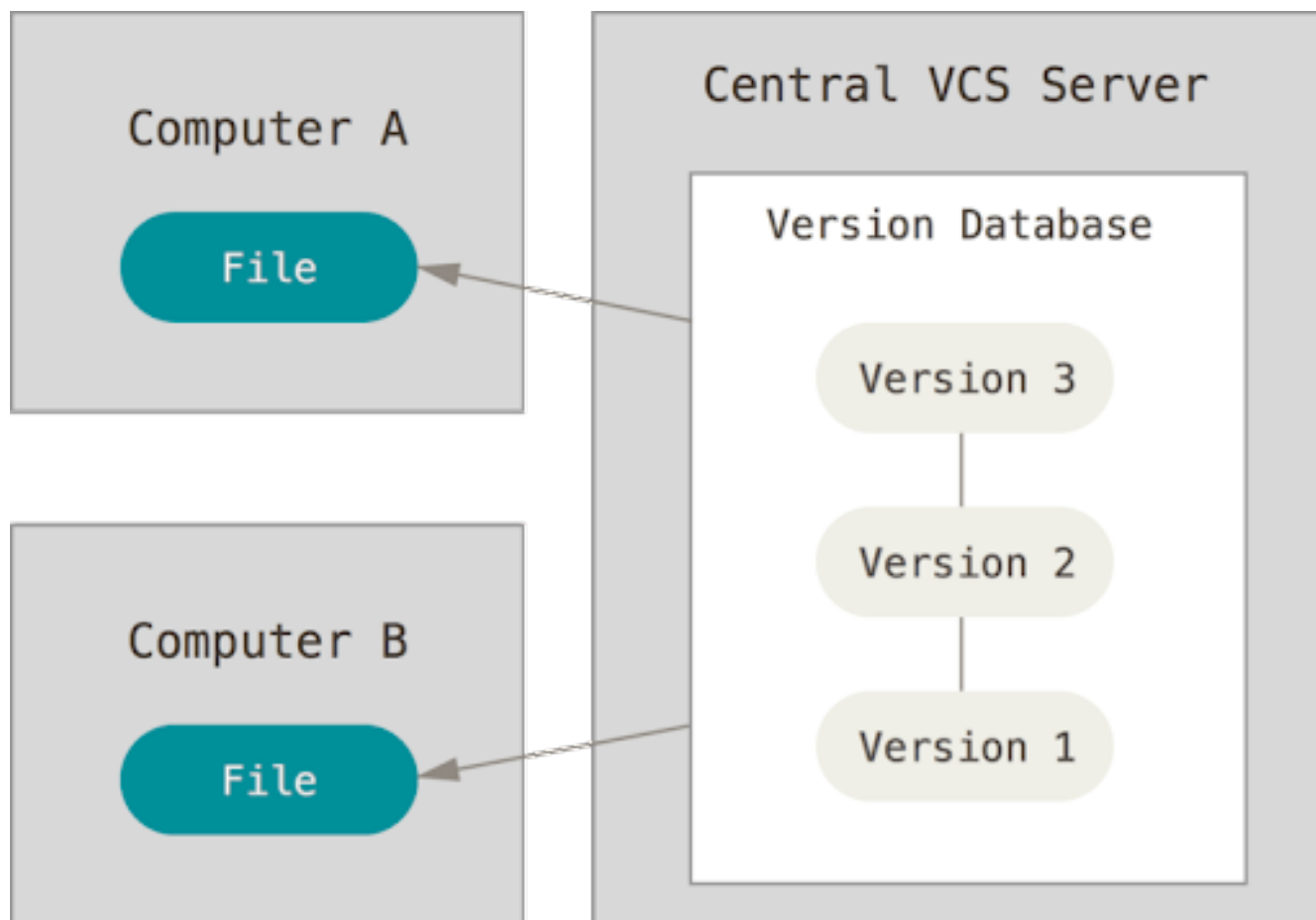


Рис. 18. Централизованный контроль версий

времени никто не сможет использовать контроль версий для сохранения изменений, над которыми работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные точки сохранения отсутствуют, о будет потеряно все – вся история проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков.

**Распределённые системы контроля версий.** В РСКВ (таких как Git, Mercurial, Bazaar или Darcs) клиенты не просто скачивают снимок всех файлов (состояние файлов на определенный момент времени) – они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, будет недоступен, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным вариантом всех данных, см. рис. 19.

Многие РСКВ могут одновременно взаимодействовать с несколькими удаленными репозиториями, благодаря этому можно работать с различными группами людей, применяя различные подходы одновременно в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

Системы контроля версий также различаются по способу разрешения конфликтов при изменении одного и того же файла со стороны нескольких разработчиков:

- механизм блокировки,
- сценарий слияния изменений перед коммитом,
- сценарий слияния изменений после коммита.

При **блокировке** рабочие файлы обычно доступны только для чтения, поэтому их нельзя просто так изменить. Необходимо дать запрос СКВ сделать рабочий файл доступным для записи, заблокировав его; только один пользователь может делать это в любой момент времени. Когда изменения будут зарегистрированы, то данные разблокируются, и рабочий файл снова становится доступным только для чтения. Это позволяет другим пользователям заблокировать файл для внесения дальнейших изменений.

В **сценарии слияния перед коммитом** СКВ отслеживает, когда осуществляется попытка выполнить коммит файла или файлов, которые уже изменились с момента начала редактирования, и в результате система тре-



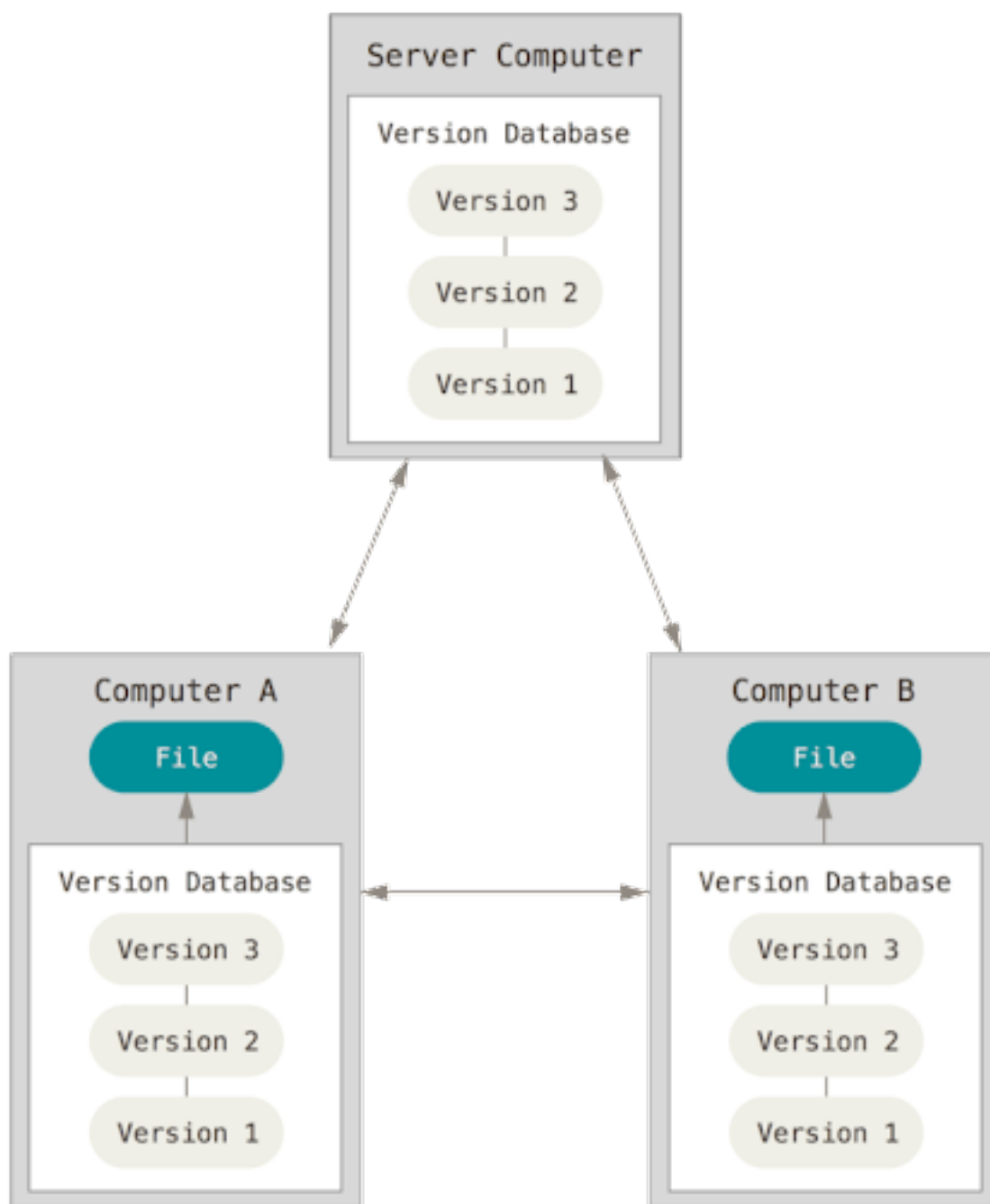


Рис. 19. Распределенный контроль версий

бует разрешения конфликта, прежде чем коммит может быть завершен.

Существуют СКВ, которые никогда не блокируют коммиты – это **сценарий слияния после коммита**. Если копия репозитория изменилась с момента извлечения файлов, коммит можно просто перенаправить в новую ветку. Впоследствии ветви могут оставаться отдельными; или любой разработчик может выполнить слияние, которое снова объединит их.

Модель «фиксация перед слиянием» приводит к очень гибкому стилю разработки, в котором разработчики создают и повторно объединяют множество небольших веток. Это упрощает экспериментирование и записывает все, что пробуют разработчики, таким образом, чтобы это было полезно для проверки кода.

В наиболее общем случае репозиторий под управлением СКВ, работающей по сценарию слияния после коммита, может иметь форму произвольно сложного ориентированного ациклического графа. Такие СКВ часто выполняют слияние с учетом истории, используя алгоритмы, которые пытаются учитывать не только содержимое сливаемых версий, но и содержимое их общих предков в графе репозитория.

## 6.1. История развития систем контроля версий

На рис. 20 показана временная шкала эволюции систем контроля версий.

В табл. 2 представлены три поколения СКВ.

Таблица 2. Поколения систем контроля версий

Поколение	Модель взаимодействия	Единица операции	Разрешение конфликтов	Примеры
1	Локальный доступ	Файл	Блокировка	RCS, SCCS
2	Централизованное	Файл / множество файлов	Слияние до коммита	CVS, SourceSafe, Subversion, Team Foundation Server

Поколение	Модель взаимодействия	Единица операции	Разрешение конфликтов	Примеры
3	Распределенное	Множество файлов	Слияние после коммита	Bazaar, Git, Mercurial, Fossil

## 6.2. Система Git

Git напоминает миниатюрную файловую систему с утилитами, надстроенными над ней, нежели просто на СКВ.

Для работы большинства операций в Git достаточно локальных файлов и ресурсов – в основном, системе не нужна никакая информация с других компьютеров в вашей сети. Так как вся история проекта хранится прямо на локальном диске, большинство операций кажутся чуть ли не мгновенными.

В Git для всего вычисляется хеш-сумма, и только потом происходит сохранение. В дальнейшем обращение к сохраненным объектам происходит по этой хеш-сумме. Это значит, что невозможно изменить содержимое файла или директории так, чтобы Git не узнал об этом. Данная функциональность встроена в Git на низком уровне и является неотъемлемой частью его философии. Вы не потеряете информацию во время её передачи и не получите повреждённый файл без ведома Git.

Механизм, которым пользуется Git при вычислении хеш-сумм, называется SHA-1 хеш. Это строка длиной в 40 шестнадцатеричных символов (0-9 и a-f), она вычисляется на основе содержимого файла или структуры каталога. SHA-1 хеш выглядит примерно так:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Вы будете постоянно встречать хеши в Git, потому что он использует их повсеместно. На самом деле, Git сохраняет все объекты в свою базу данных не по имени, а по хеш-сумме содержимого объекта.

Когда вы производите какие-либо действия в Git, практически все из них только добавляют новые данные в базу Git. Очень сложно заставить систему удалить данные либо сделать что-то, что нельзя впоследствии отменить. Как и в любой другой СКВ, вы можете потерять или испортить свои изменения, пока они не зафиксированы, но после того, как вы зафиксируете снимок

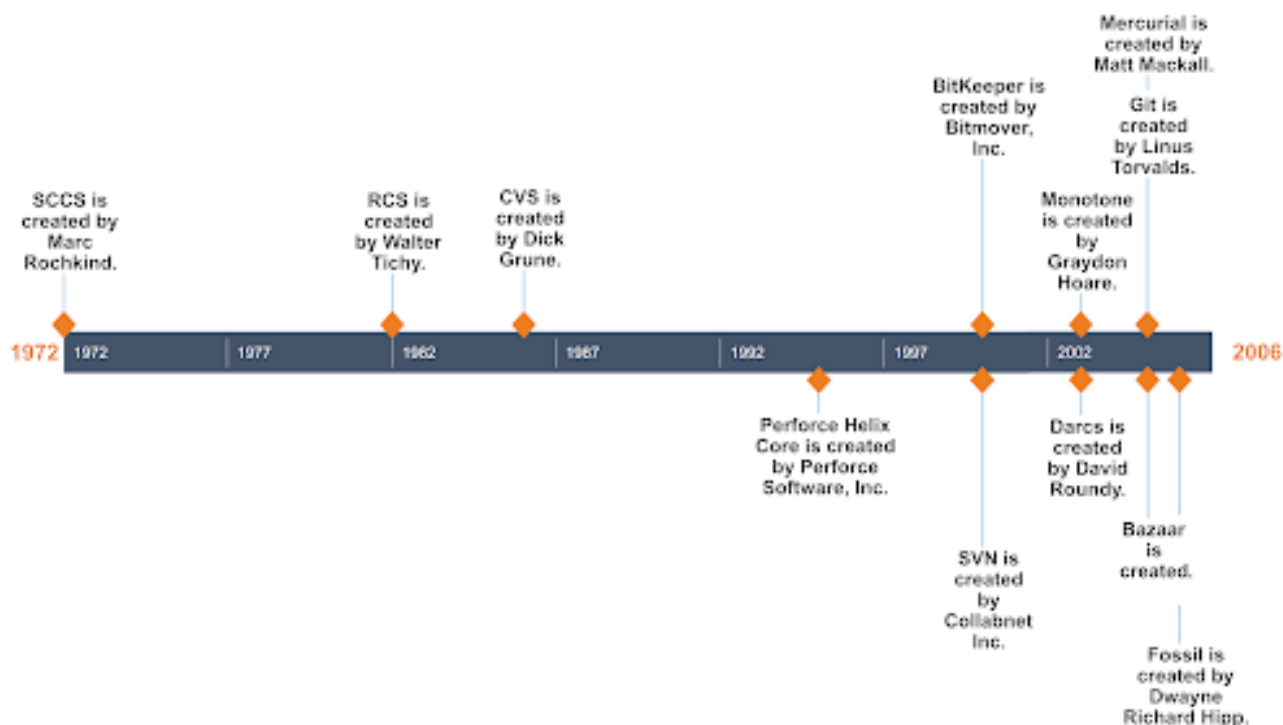


Рис. 20. Эволюция систем контроля версий

в Git, будет очень сложно что-либо потерять, особенно, если вы регулярно синхронизируете свою базу с другим репозиторием.

У Git есть три основных состояния, в которых могут находиться ваши файлы: зафиксированное (committed), изменённое (modified) и подготовленное (staged).

- Зафиксированный значит, что файл уже сохранён в вашей локальной базе.
- К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы.
- Подготовленные файлы – это изменённые файлы, отмеченные для включения в следующий коммит.

Существует три основных элемента проекта Git:

- Git-директория (Git directory),
- рабочая директория (working directory),
- область подготовленных файлов (staging area).

**Git-директория** – это то место, где Git хранит метаданные и базу объектов вашего проекта. Это самая важная часть Git, и это та часть, которая копируется при клонировании репозитория с другого компьютера.

**Рабочая директория** является снимком версии проекта. Файлы распаковываются из сжатой базы данных в Git-директории и располагаются на диске, для того чтобы их можно было изменять и использовать.

**Область подготовленных файлов** – это файл, обычно располагающийся в вашей Git-директории, в нём содержится информация о том, какие изменения попадут в следующий коммит. Эту область ещё называют «индекс», однако называть её stage-область также общепринято.

Базовый подход в работе с Git выглядит так:

- Вы изменяете файлы в вашей рабочей директории.
- Вы выборочно добавляете в индекс только те изменения, которые должны попасть в следующий коммит, добавляя тем самым снимки только этих изменений в область подготовленных файлов.
- Когда вы делаете коммит, используются файлы из индекса как есть, и этот снимок сохраняется в вашу Git-директорию.

Если определенная версия файла есть в Git-директории, эта версия считается зафиксированной. Если версия файла изменена и добавлена в индекс, значит, она подготовлена. И если файл был изменен с момента последнего распаковывания из репозитория, но не был добавлен в индекс, он считается измененным.

## 6.3. Модель данных Git

**Снимки.** Git моделирует историю файлов и каталогов в некотором каталоге верхнего уровня как серию снимков. В терминологии Git файл называется «blob» (Binary Large Object), это просто набор байтов. Каталог называется «деревом», и он сопоставляет имена с blobs или деревьями (то есть каталоги могут содержать в себе другие каталоги).

Снимок - это отслеживаемое СКВ дерево.

Пример дерева:

```
<root> (tree)
|
+- foo (tree)
|  |
|  + bar.txt (blob, contents = "hello world")
|
+- baz.txt (blob, contents = "git is wonderful")
```

Модель истории. В Git **история** - это ориентированный ациклический граф снимков. В псевдокоде:

```
// файл - последовательность байт
type blob = array<byte>

// директория содержит именованные файлы или директории
type tree = map<string, tree | blob>

// коммит содержит родителей, метаданные и дерево
type commit = struct {
    parent: array<commit>
    author: string
    message: string
    snapshot: tree
}
```

**Объекты.** Объект это blob, дерево или коммит. Объекты адресуются по своим хеш-значениям SHA-1.

```
type object = blob | tree | commit

objects = map<string, object>

def store(object):
    id = sha1(object)
    objects[id] = object

def load(id):
    return objects[id]
```

**Ссылки.** Ссылки указывают на объекты.

```
references = map<string, string>

def update_reference(name, id):
    references[name] = id

def read_reference(name):
    return references[name]

def load_reference(name_or_id):
```

```
if name_or_id in references:
    return load(references[name_or_id])
else:
    return load(name_or_id)
```

Управление git удобнее всего осуществлять из командной строки. Ниже представлены некоторые команды git:

#### Базовые команды

```
git help <command>: помощь по git
git init: инициализация нового репозитория
git status: текущий статус репозитория
git add <filename>: добавить файл в подготовительную область
git commit: создать новый коммит
git log: показать историю изменений
git log --all --graph --decorate: история, как граф
git diff <filename>: изменения по сравнению с состоянием в
    ↪ подготовительной области
git diff <revision> <filename>: изменения в разных снимках для
    ↪ файла
git checkout <revision>: обновление HEAD и текущей ветки
```

#### Ветки и слияние

```
git branch: показать ветки
git branch <name>: создать ветку
git checkout -b <name>: создать ветку и переключиться на нее
то же, что и git branch <name>; git checkout <name>
git merge <revision>: слияние в текущую ветку
```

#### Удаленная работа

```
git remote: список удаленных репозиториях
git remote add <name> <url>: добавить удаленный репозиторий
git push <remote> <local branch>:<remote branch>: отправить данные
    ↪ для изменения в удаленном репозитории
git branch --set-upstream-to=<remote>/<remote branch>: установить
    ↪ взаимосвязь между локальной и удаленной ветками
git fetch: получить данные из удаленного репозитория
git pull: то же, что и git fetch; git merge
git clone: загрузить удаленный репозиторий
```

#### Отмена действий

`git commit --amend`: отредактировать содержимое коммита  
`git reset HEAD <file>`: отменить добавление файла



# 7. Документация как код

Как известно, большинство программистов не любит писать документацию к своим проектам. На это есть причины. В частности, документацию трудно поддерживать в актуальном состоянии в процессе разработки программы. Кроме того, традиционный подход к ведению технической документации с использованием редакторов в духе Microsoft Word с точки зрения разработчика сильно отличается от процессов ведения программного проекта.

В связи с вышесказанным перспективным является подход «документация как код» (docs as code), основная идея которого в использовании для создания технической документации тех же процессов, что и для разработки программ. Подход «документация как код» отличается следующими особенностями:

- Использование текстовых языков разметки, удобных как для чтения человеком, так и с точки зрения машинной обработки.
- Использование текстовых языков описания графических материалов.
- Использование системы контроля версий для хранения проекта документации.
- Использование инструментов командной строки для автоматической проверки, сборки документации и непрерывной интеграции (continuous integration).
- Ориентация на выходной web-формат.

## 7.1. Языки разметки

Языки разметки, помимо очевидной возможности написания текстов, поддерживают специальные команды, отвечающие за внешний вид и структурные особенности документа. В отличие от обычных WYSIWYG-редакторов («что вижу на экране, то и получу в документе») язык разметки

позволяет документ «запрограммировать», при этом «программа» на языке разметки и ее результат в виде документа отличаются друг от друга.

Очевидным примером языка разметки является HTML, но для задач составления документации было создано множество специальных языков, в частности:

- LaTeX,
- Markdown,
- reStructuredText,
- AsciiDoc.

Важным достоинством языка разметки является удобство использование системы контроля версий – в истории репозитория легко отследить изменения, внесенные в документ. Этого не удалось бы добиться с двоичными форматами в духе docx.

Одной из важнейших проблем проектирования языка разметки является обеспечение необходимой гибкости в компьютерной верстке документа при использовании облегченного, почти «невидимого» для пользователя командного языка.

Один из древнейших и, пожалуй, самый мощный язык разметки – TeX, который используется в одноименной системе компьютерной верстки. TeX был разработан Д. Кнудом в 1978 году для задач написания литературы в области компьютерных наук. В 1984 году Л. Лэмпорт создал набор макрорасширений для TeX под названием LaTeX. Сегодня LaTeX используется для написания статей в журналах по математике и физике, создания технических книг, дипломов и диссертаций.

LaTeX отличают средства автоматизации создания документов, это касается, в частности, построения списка литературы, нумерации элементов и ссылок на них, оптимизации размещения элементов на страницах и описания математических формул.

Ниже представлен пример простого документа в LaTeX:

```
\documentclass[14pt]{article} % Формат страниц
\usepackage{polyglossia} % Поддержка русского языка
\setmainlanguage{russian}
\setmainfont{Times New Roman} % Настройка шрифта

\title{Тестовый документ} % Заголовок
\author{П.Н. Советов} % Автор
```

```
\date{\today} % Дата создания
```

```
\begin{document} % Тело документа
```

```
\maketitle % Вставка заголовка
```

Это простой `\textbf{пример}` документа в `\LaTeX`.

```
\end{document}
```

Можно заметить, что команды в `LaTeX` предваряются символом `\` и могут иметь аргументы, заключенные в скобки различных форм.

Результат компиляции документа с помощью `xelatex` показан на рис. 21.

## Тестовый документ

П.Н. Советов

5 ноября 2021 г.

Это простой **пример** документа в `LaTeX`.

Рис. 21. Результат компиляции `LaTeX`-документа

Особый интерес представляет использующийся в `LaTeX` язык разметки математических формул. Формулы обрамляются символами `$` (встраивание в текст) или `$$` (в отдельной строке). Примеры простых формул представлены ниже:

```
$$
```

```
a^n + b^n = c^n
```

```
$$
```

```
$$
```

```
x_{1,2}=\frac{-b \pm \sqrt {b^2-4ac}}{2a}
```

```
$$
```

```

$$
f(x) = \frac{1}{\sigma \sqrt{2\pi}}
      e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}
$$

$$
\eta(T|a) = \sum_{v \in \text{vals}(a)} \left\{ \frac{|S_a(v)|}{|T|} \right.
      \left. \cdot \eta\left(S_a\left(\left(v\right)\right)\right) \right\}
$$

```

Результат компиляции формул представлен на рис. 22.

$$a^n + b^n = c^n$$

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$f(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

$$\eta(T|a) = \sum_{v \in \text{vals}(a)} \frac{|S_a(v)|}{|T|} \cdot \eta(S_a(v))$$

Рис. 22. Полученные формулы

Знакомство с языком описания формул LaTeX очень полезно, поскольку этот язык или его подмножества используются во многих современных системах, например, в языке разметки Wikipedia.

## 7.2. Грамотное программирование

Как уже говорилось выше, особенную проблему представляет разделенность программного кода и документации на программный проект. Ранней попыткой решить указанную проблему является подход «грамотного программирования» (literate programming) Д. Кнута в виде системы WEB, созданной в 1984 году. WEB основана на системе TeX и позволяет вести документацию с внедренным в нее программным кодом.

С помощью инструмента **weave** из WEB-файла извлекается только часть, связанная с документацией. С помощью инструмента **tangle** из WEB-файла извлекается программный код. Схема работы WEB показана на рис. 23.

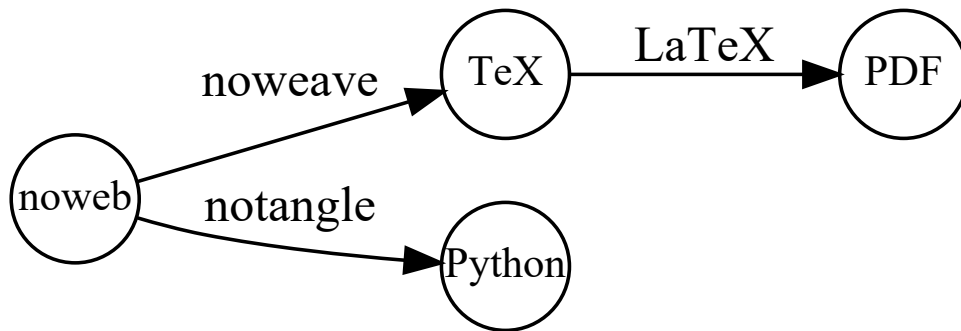


Рис. 23. Схема работы системы WEB

На рис. 24 показан пример LaTeX-документа, извлеченного из WEB-файла.

### Вставка в бинарное дерево

В этом разделе описывается реализация функции `insert` для вставки элемента в бинарное дерево.

```

<insert.py>≡
def insert(tree, key):
    <Случай, когда узел — None>
    <Случай, когда ключ-аргумент не равен ключу узла>

    Рассмотрим случай, когда узел представляет собой None.
    <Случай, когда узел — None>≡
    if not tree:
        tree = Node(key)

    Теперь рассмотрим случаи, когда ключ-аргумент не равен ключу узла.
    <Случай, когда ключ-аргумент не равен ключу узла>≡
    elif key < tree.key:
        tree = Node(tree.key, insert(tree.left, key), tree.right)
    elif key > tree.key:
        tree = Node(tree.key, tree.left, insert(tree.right, key))

    Наконец, рассмотрим оставшийся вариант.
    <insert.py>+≡
    return tree

    Все.
  
```

Рис. 24. Пример WEB-документа

Из приведенного WEB-документа может быть извлечен также следующий код:

```

def insert(tree, key):
    if not tree:
        tree = Node(key)
    elif key < tree.key:
        tree = Node(tree.key, insert(tree.left, key), tree.right)
    elif key > tree.key:
  
```

```

    tree = Node(tree.key, tree.left, insert(tree.right, key))
    return tree

```

В какой-то мере черты грамотного программирования унаследовала система Jupyter-блокнотов, в которой документы представлены в виде последовательности ячеек. Ячейка может либо содержать программный код, либо – документацию. Jupyter-блокноты используются, в основном, в области научно-технических расчетов и для анализа данных.

## 7.3. Markdown и Pandoc

Одним из простейших языков разметки является Markdown. Его авторы преследовали цель получения незаметного командного языка, чтобы файлы на Markdown было легко читать, как есть, даже без трансляции в какое-то выходное представление.

Ниже показаны примеры элементов синтаксиса Markdown:

```

# Заголовок 1
## Заголовок 2
## Заголовок 3

```

Параграф с текстом и *[ссылкой]* (<https://pandoc.org/>).

Параграф с текстом, выделенным **\*\*жирным\*\*** и *\*курсивом\**.

> Важная цитата (Автор)

Список элементов:

1. Первый.
1. Второй.
  - \* Вложенный первый.
  - \* Вложенный второй.
1. Третий.

Поле 1	Поле 2	
-----	-----	
Данные 1	Данные 3	
Данные 2	Данные 4	

```
```python
{
  "name": "Ivan",
  "last_name": "Drago",
  "age": 25
}
...`
```

Markdown, в силу простоты и читаемости своего синтаксиса, представляет собой возможную альтернативу LaTeX. Однако, при использовании более-менее сложного форматирования документов возможностей Markdown быстро перестанет хватать. В этой ситуации может помочь инструмент Pandoc, предназначенный для трансляции документов из одного представления в другое. Особенность Pandoc в том, что в нем поддерживается расширенный вариант Markdown, который, в свою очередь, можно дополнить рядом сторонних модулей.

Архитектура Pandoc показана на рис. 25. Работу Pandoc можно разбить на три этапа:

1. Одно из входных представлений с помощью поддерживаемых трансляторов для чтения (readers) преобразуется во внутреннее представление Pandoc – дерево абстрактного синтаксиса.
2. На уровне AST могут бы применены пользовательские фильтры (filters).
3. Полученное AST с помощью поддерживаемых трансляторов для записи (writers) преобразуется в одно из выходных представлений.

В варианте markdown от Pandoc поддерживаются некоторые элементы LaTeX, в частности, язык описания математических формул.

## 7.4. Языки описания диаграмм

Как и в случае текста, языки описания графических материалов позволяют легко вносить и отслеживать изменения в рисунок. Еще одним преимуществом таких языков является возможность автоматизации построения рисунков. Это касается, в частности, построения различных диаграмм на основе автоматического анализа классов и модулей из программного кода.

Одним из наиболее популярных инструментов в этой области является Graphviz, в котором для описания графов различного вида используется язык Dot.

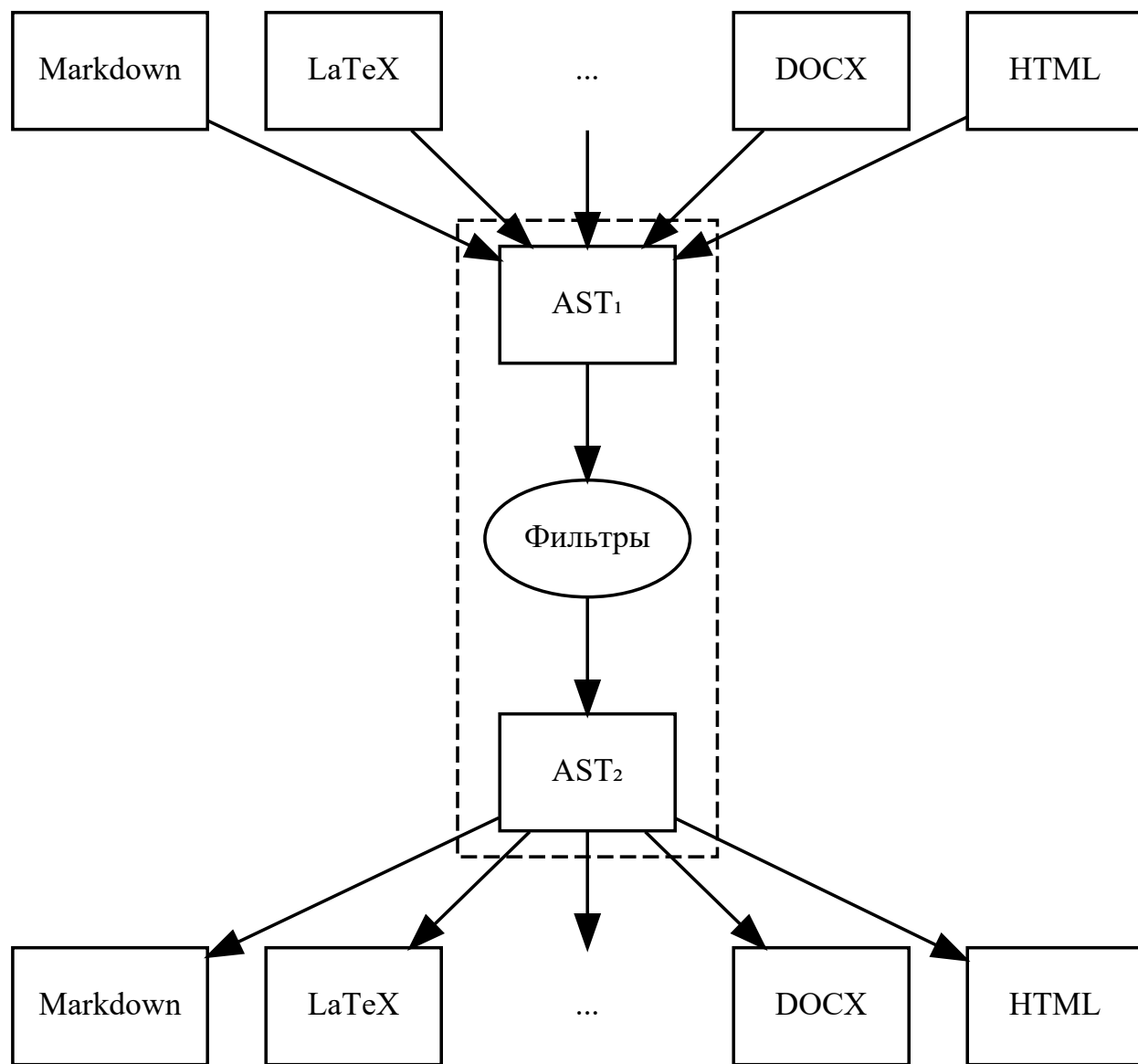


Рис. 25. Архитектура Pandoc



Пример кода на языке Dot показан далее:

```
digraph G {  
  n1 [style=filled, color=brown1, label="1", shape=oval]  
  n2 [style=filled, color=darkolivegreen1, label="2", shape=box]  
  n3 [style=filled, color=aquamarine, label="3", shape=circle]  
  
  n1 -> n2 -> n3  
  n3 -> n1  
}
```

Результат компиляции в графический файл представлен на рис. 26.

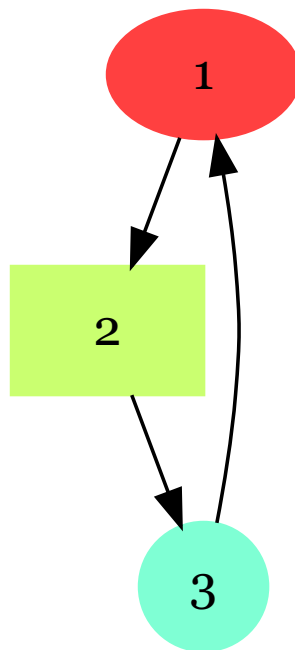


Рис. 26. Результат работы Graphviz

Еще одним популярным инструментом является PlantUML, предназначенный для создания как UML-диаграмм различного вида, так и для диаграмм иного вида (диаграммы Ганта, интеллект-карты и проч.).

Ниже представлен пример диаграммы, описанной на языке PlantUML:

```
@startuml  
skinparam monochrome true  
skinparam shadowing false
```

A -> B: шаг

```

activate B
B -> C: шаг

activate C
C --> C: действие
C -> B: шаг
deactivate C

B -> A: шаг
deactivate B
@enduml

```

Результат компиляции в графический файл представлен на рис. 27.

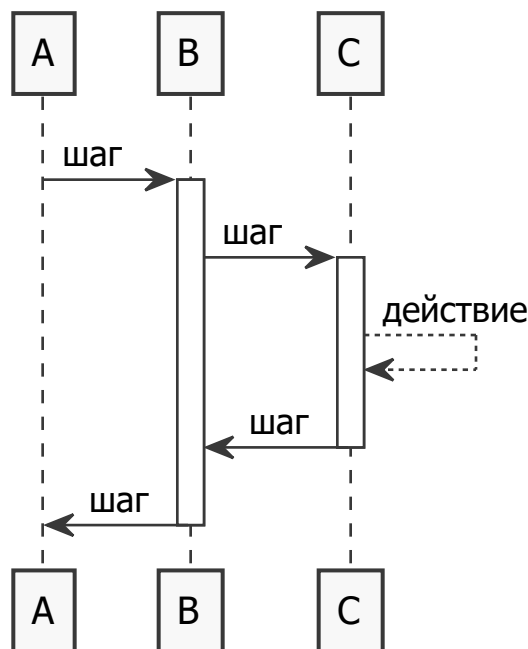


Рис. 27. Результат работы PlantUML

## 7.5. Генераторы документации на основе исходных текстов

Очевидным способом объединения документации и программного кода является подробное комментирование программного кода. При внесении изменений в программу имеет возможность сразу же обновить документирующие поведение программы комментарии в коде. Существуют инструменты, позволяющие автоматически извлечь из файла программы специальным об-

разом оформленные комментарии и оформить результат в виде справочной документации по программному модулю.

Примеры генераторов документации:

- Javadoc для языка Java.
- Doxygen для C++ и некоторых других языков.
- Расширения для системы Sphinx, поддерживающие целый ряд языков.

Ниже показан пример программы на языке C со специальными комментариями, поддерживаемыми системой Doxygen. Обратите внимание на специальные ключевые слова `\file`, `\brief` и `\param`:

```
/// \file main.cpp
/// Модуль main.

#include <stdio.h>

/// \brief Главная функция.
/// \param int argc Счетчик аргументов.
/// \param char **argv Указатель на аргументы.
int main(int argc, char **argv) {
    return 0;
}
```

Далее приведен пример программы на языке Питон, с комментариями, поддерживаемыми системой Sphinx. Обратите внимание на специальные ключевые слова `param`, `type` и `return`:

```
"""
Модуль main.
"""

def main(x, y):
    """
    Функция main.

    :param x: Параметр x.
    :type x: str
    :param y: Параметр y.
    :type y: int
    :return: Ничего не возвращает.
    """
```

# 8. Виртуализация

## 8.1. Что такое виртуализация

Определение слова «виртуальный» можно найти, к примеру, в толковом словаре Ожегова:

«ВИРТУАЛЬНЫЙ, -ая, -ое; -лен, -льна (спец.). Несуществующий, невозможный. Виртуальные миры. Виртуальная реальность (несуществующая, воображаемая). В. образ (в компьютерных играх)»

Некоторые сюжетные элементы на тему виртуальности из мира фантастики вполне встречаются и в компьютерном мире. Возьмем ситуацию, когда герой обнаруживает внутри виртуального мира еще один, внутренний виртуальный мир. Это похоже на запуск в браузере ОС Linux, в которой, в свою очередь, запускается программа DOSBox, имитирующая работу старого компьютера под управлением MS-DOS.

Можно вспомнить еще одну типичную для фантастических произведений ситуацию, когда герой пытается понять, в реальном ли он находится мире, или же – в виртуальном. Подобная проблема возникает, к примеру, для выполняемой компьютерной программы, которая пытается определить, не происходит ли нежелательный анализ ее поведения под управлением виртуальной машины.

В случае компьютерной системы «реальным миром» является хост-система, а «виртуальным миром» – виртуальная машина, гостевая система или, в более общем смысле, некоторый виртуальный ресурс.

**Виртуализацией** будем называть создание виртуальной (абстрактной, имитируемой) версии ресурса компьютерной системы. Для виртуализации типичными являются следующие свойства:

- изолированность виртуального ресурса от хост-системы и от других

- виртуальных ресурсов,
- разделение ресурсов хост-системы между виртуальными ресурсами.

На рис. 28 показаны основные виды виртуализации.

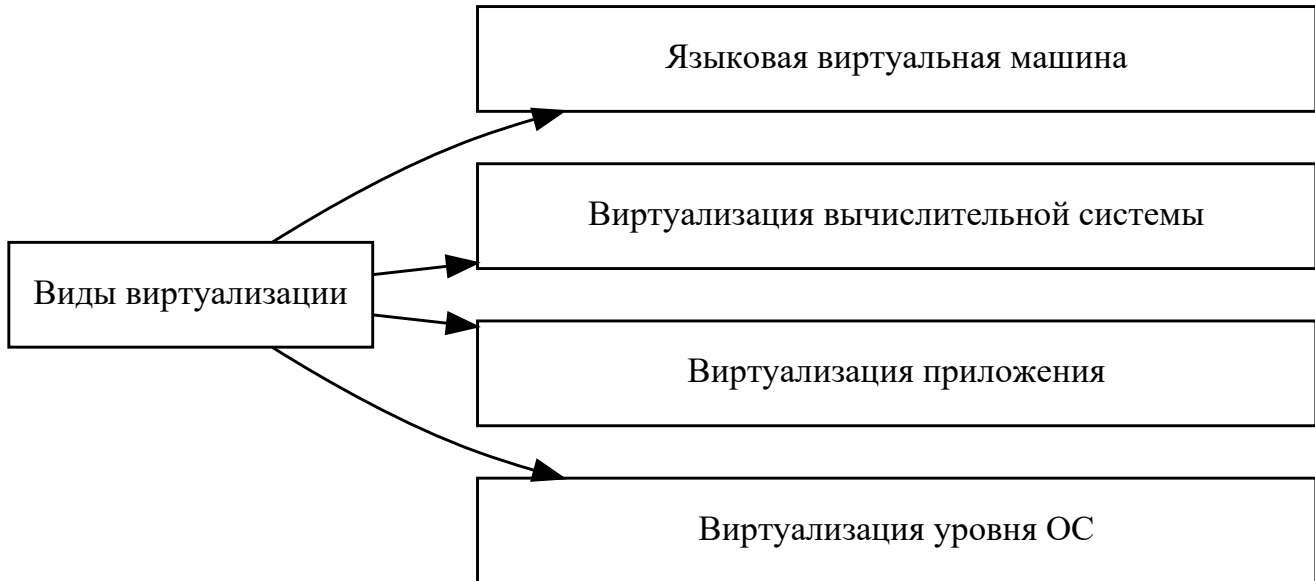


Рис. 28. Виды виртуализации

Понятие виртуальной машины связано с самим понятием алгоритма, определяющего процесс вычислений, который может быть выполнен на машине Тьюринга или на каком-то ином абстрактном вычислителе, полном по Тьюрингу. Доказательство полноты по Тьюрингу достигается демонстрацией реализации внутри интересующего нас вычислителя виртуальной машины, имитирующей работу какого-либо из вычислителей, Тьюринг-полнота которого известна.

На рис. 29 представлены основные техники обеспечения виртуализации.

## 8.2. Языковые виртуальные машины

Языковые виртуальные машины предназначены для исполнения программ на конкретном языке программирования (или семействе таких языков) в условиях различных программно-аппаратных платформах без перекомпиляции. Иными словами, упрощается портирование программ.

Рассмотрим классический пример П-кода (P-code) для виртуального выполнения программ на языке Паскаль, предложенный в середине 70-х годов. Специальный вариант компилятора Паскаля порождает платформонезави-

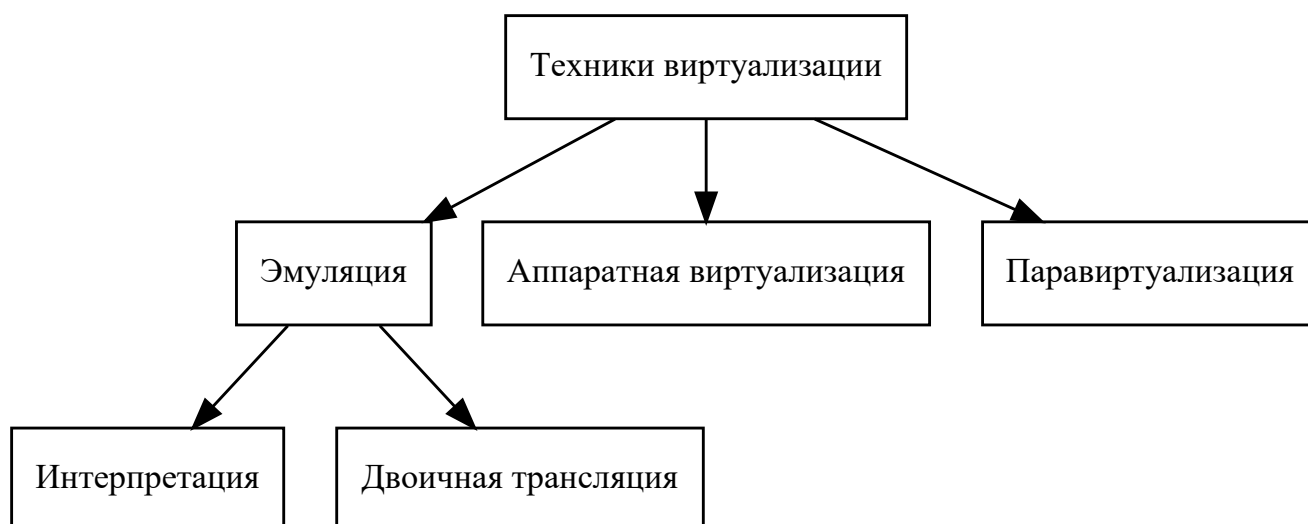


Рис. 29. Техники виртуализации

симый П-код. Для различных программно-аппаратных платформ реализованы интерпретаторы П-кода, также включающие библиотеку времени выполнения. В результате компилятор существует только в одном варианте и для каждой из новых целевых платформ достаточно реализовать небольшую программу – интерпретатор П-кода.

К ярким историческим примерам успешного портирования компьютерных игр на множество различных игровых платформ относятся текстовая игра Zork (1979), реализованная в коде виртуальной Z-машины, а также игра Another World (1991), имеющая изощренную виртуальную машину с поддержкой многопоточности, графики и звука.

К популярным современным языковым виртуальным машинам можно отнести:

- Java Virtual Machine (JVM). Виртуальная машина для Java.
- Common Language Runtime (CLR). Виртуальная машина среды .NET.
- CPython Virtual Machine. Виртуальная машина языка Python.
- WebAssembly (WASM). Виртуальная машина для веб-приложений.

**Языковая виртуальная машина** выполняет команды абстрактного процессора, ориентированного на конструкции конкретного языка или целого семейства языков. Программу в таком представлении принято называть байткодом. Такое название закрепилось исторически, оно связано с виртуальной машиной языка Smalltalk, в которой большинство команд кодировалось одним байтом.

Поскольку реальные процессоры, в большинстве случаев, не поддержива-

ют на аппаратном уровне исполнение **байткода**, то используется программная модель языковой машины.

Архитектуры виртуальных машин, как и архитектуры реальных процессоров, можно разделить на два класса:

- стековые машины,
- регистровые машины.

Рассмотрим байткод различных виртуальных машин для вычисления выражения

$$bb - 4ac.$$

В JVM используется стековая модель вычислений:

```
0: iload_1
1: iload_1
2: imul
3: iconst_4
4: iload_0
5: imul
6: iload_2
7: imul
8: isub
9: ireturn
```

В CPython также используется стековая модель вычислений:

```
0 LOAD_FAST
2 LOAD_FAST
4 BINARY_MULTIPLY
6 LOAD_CONST
8 LOAD_FAST
10 BINARY_MULTIPLY
12 LOAD_FAST
14 BINARY_MULTIPLY
16 BINARY_SUBTRACT
18 RETURN_VALUE
```

Виртуальная машина языка Lua использует регистровую вычислительную модель (см. числа после имен операций):

```

1  MUL      3 1 1
2  MMBIN    1 1 8   ; __mul
3  MULK     4 0 0   ; 4
4  MMBINK   0 0 8 1 ; __mul 4 flip
5  MUL      4 4 2
6  MMBIN    4 2 8   ; __mul
7  SUB      3 3 4
8  MMBIN    3 4 7   ; __sub
9  RETURN1  3
10 RETURN0

```

Во многих случаях для выполнения байткода быстрогодействия интерпретатора оказывается недостаточно. В этой связи может использоваться трансляция. Различают следующие варианты трансляции:

- АОТ (Ahead-of-Time). Полная трансляция байткода в машинный код целевой платформы перед ее запуском.
- JIT (Just-in-Time). Динамическая трансляция областей программы прямо во время интерпретации байткода.

### 8.3. Виртуализация вычислительной системы

Устаревание программного и аппаратного обеспечения является серьезной проблемой в области информационных технологий. Здесь помощь может оказать виртуализация. Устаревшее оборудование, к которому уже не найти современных драйверов, может быть заменено своей виртуальной программной версией. Программа, предназначенная для выполнения на редкой или устаревшей вычислительной системе, может быть выполнена в рамках виртуальной машины уже на современном компьютере. Виртуализация также полезна для экономного управления вычислительными ресурсами. С использованием виртуализации пользователь на одном физическом компьютере может работать с несколькими виртуальными компьютерами, имеющим различные характеристики и использующими различные ОС. Этот подход используется для организации виртуальных серверов для веб-хостинга, а также применяется в сфере облачных вычислений.

Виртуализация уровня вычислительной системы имеет давнюю историю. Еще в конце 60-х годов компания IBM реализовала в своей ОС CP/CMS поддержку виртуализации на аппаратном уровне. Таким образом было организована работа с несколькими программными версиями компьютера System/360-370. С массовым развитием персональных компьютеров интерес



к виртуализации сильно угас. Возобновление этого интереса произошло уже в 1999 году, когда компания VMware выпустила коммерчески успешное ПО VMware Workstation для виртуализации работы ОС Linux и Windows на компьютерах с архитектурой x86.

Для организации виртуальной машины требуется специальный программный модуль – монитор виртуальных машин или гипервизор. **Гипервизор** управляет доступом виртуальных машин к физическим ресурсам хост-машины. В этом отношении гипервизор ведет себя аналогично ядру операционной системы. Гипервизоры разделяют на следующие два типа:

1. Работает прямо на оборудовании хост-машины, без участия ОС.
2. Работает в рамках хост-ОС и, возможно, использует модуль ядра для управления физическими ресурсами хост-машины.

Теоретические основы виртуализуемости, определяющие, насколько эффективно работает гипервизор, были предложены в 1974 году Ж. Попеком (G. Popek) и Р. Голдбергом (R. Goldberg). Рассмотрим подробнее суть предложений этих авторов.

Выделены следующие типы команд компьютера:

- Привилегированные. Выполнение которых вызывает исключение, если это выполнение происходит вне режима ядра ОС.
- Чувствительные. Команды, которые позволяют управлять состоянием компьютера или отслеживать это состояние.
- Безвредные. Остальные команды.

Сформулированы следующие свойства, которыми должен обладать гипервизор:

- Безвредные команды выполняются аппаратурой напрямую, без привлечения гипервизора.
- Программы влияют на доступные им ресурсы системы только опосредованно, с помощью гипервизора.
- Программа под управлением гипервизора выполняется точно так же, как и без него, за исключением временных характеристик и объема доступных ресурсов.

Наконец, авторы сформулировали следующую **теорему виртуализуемости**: построение гипервизора, удовлетворяющего этим свойствам для заданного компьютера возможно, если множество чувствительных команд этого компьютера является подмножеством привилегированных команд.

Аппаратная виртуализация, основанная на обработке гипервизором исключений со стороны привилегированных команд использовалась еще в старых ОС от компании IBM. Тем не менее, для персональных компьютеров архитектуры x86 долгое время представленная выше теорема не выполнялась. В начале 2000-х реализация гипервизора на архитектуре x86-64 требовала серьезных ухищрений. Использовалась, в частности, динамическая трансляция программы с целью замены чувствительных команд на соответствующие обращения к гипервизору. Кроме того, применялась техника **паравиртуализации**, при использовании которой в гостевую ОС должны быть внесены изменения, включающие обращения к гипервизору. Наконец, в середине 2000-х компании Intel и AMD добавили в свои процессоры аппаратную поддержку виртуализации. Современные процессоры поддерживают аппаратную виртуализацию как процессора, так и ОЗУ, а также устройств ввода-вывода.

При использовании виртуальной машины с отличающейся от хост-машины процессорной архитектурой необходима **эмуляция**, которая может быть реализована в виде интерпретатора машинного кода, а также на основе статического (АОТ) или динамического (JIT) двоичного транслятора.

Одной из популярных программ для задач виртуализации вычислительных систем является QEMU. В этой программе реализована эмуляция ряда компьютеров различных архитектур с набором периферийных устройств. В QEMU поддерживается аппаратная виртуализация с использованием сторонних гипервизоров, в числе которых KVM и HAXM.

Рассмотрим пример использования QEMU. Предположим, мы хотим использовать дистрибутив Linux Alpine на виртуальной машине с архитектурой x86-64. В первую очередь создадим виртуальный жесткий диск размером 1 Гбайт, на который будет установлен дистрибутив Linux:

```
qemu-img create -f qcow2 alpine.qcow2 1G
```

Созданный жесткий диск доступен в виде файла alpine.qcow2. Теперь необходимо запустить виртуальную систему x86-64 с объемом ОЗУ в 1 Гбайт для установки Alpine на виртуальный жесткий диск. При этом используется виртуальный CD-ROM и ISO-файл alpine-standard-3.14.2-x86\_64.iso:

```
qemu-system-x86_64 -m 1G -cdrom alpine-standard-3.14.2-x86_64.iso  
↪ -hda alpine.qcow2
```

После установки Alpine на жесткий диск можно перезагрузить виртуаль-

ную систему и далее работать с ней с помощью следующей команды:

```
qemu-system-x86_64 -m 1G -hda alpine.qcow2
```

## 8.4. Виртуализация приложения

Виртуализация приложения позволяет выполнять приложения на тех платформах, для которых они не были предназначены. При этом виртуализируются интерфейсы операционной системы. Примером виртуализации уровня приложения является система Wine, позволяющая запускать приложения для Windows внутри Linux.

В QEMU имеется эмуляция режима пользователя, которая позволяет выполнять Linux-приложения, скомпилированные для одной процессорной архитектуры, на другой процессорной архитектуре.

## 8.5. Виртуализация уровня ОС

При виртуализации уровня ОС виртуализация аппаратных ресурсов обычно отсутствует. Существует единственное хост-ядро ОС и набор изолированных друг от друга пользовательских пространств, подключаемых к нему. Такие подключаемые пользовательские пространства называются **контейнерами**. Каждый контейнер содержит отдельную копию файловой системы ОС с установленным набором программ. Использование контейнеров облегчает установку, использование и взаимодействие сложных программ. В отличие от виртуализации, контейнеризация не несет накладных расходов на выполнение гостевого ядра ОС.

Рассмотрим пример использования контейнеров в системе Docker. Установим контейнер дистрибутива Alpine:

```
docker pull alpine
```

Запустим (**run**) контейнер в интерактивном режиме (**-it**), вызвав командный интерпретатор **sh** в Alpine:

```
docker run -it alpine sh
```

Необходимо учитывать, что после окончания сеанса работы с контейнером все несохраненные данные исчезнут. Данные, которые необходимо сохранять, например, файлы базы данных, записываются в каталог хост-

системы. Команда `Docker -v` указывает этот каталог, а также его представление в файловой системе контейнера (`/my_data`):

```
docker run -it -v "$(pwd)/my_data:/my_data" alpine sh
```

Для построения контейнера удобно использовать конфигурационный файл `Dockerfile`. Предположим, стоит задача построить контейнер на базе Alpine с добавленным интерпретатором языка Python. Для выполнения этой задачи `Dockerfile` имеет следующий вид:

```
FROM alpine
RUN apk add python3
```

Команда `FROM` указывает в этом примере базовый контейнер, а команда `RUN` – те команды, которые необходимо выполнить в процессе построения контейнера.