

Отчёт лабораторной работы №3

Дисциплина: Операционные системы

Касьянов Даниил Владимирович

Содержание

1	Цель работы	5
2	Задание	6
3	Выполнение лабораторной работы	7
4	Контрольные вопросы	18
5	Выводы	24

List of Tables

List of Figures

1 Цель работы

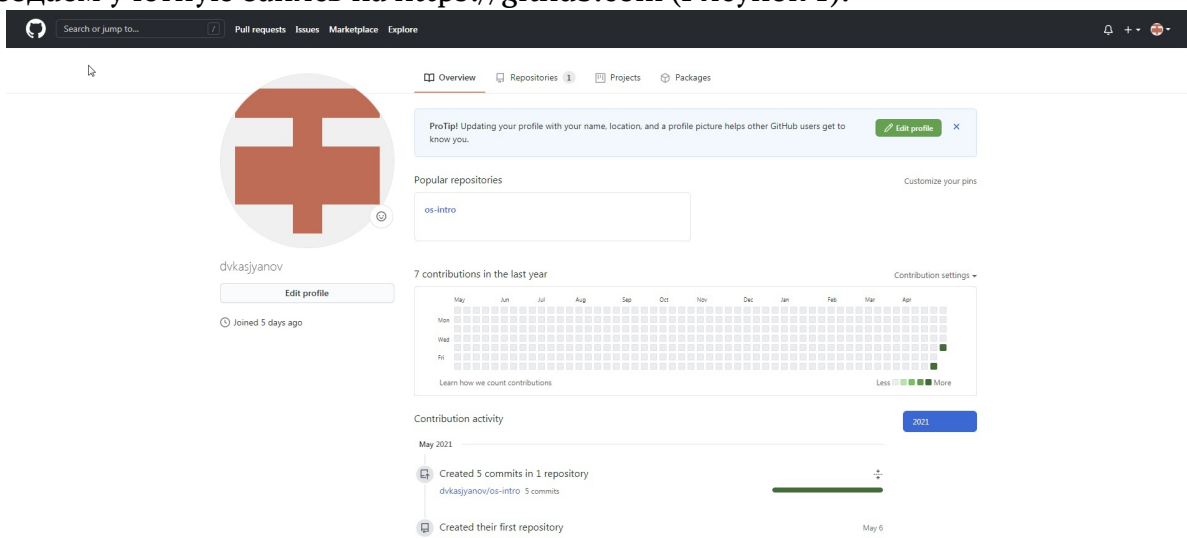
Целью данной работы является изучение идеологии применения средств контроля версий.

2 Задание

- Сделайте отчёт по предыдущей лабораторной работе в формате Markdown.
- В качестве отчёта просьба предоставить отчёты в 3 форматах: pdf, docx и md (в архиве, поскольку он должен содержать скриншоты, Makefile и т.д.).

3 Выполнение лабораторной работы

Создаем учётную запись на <https://github.com> (Рисунок 1).



(Рисунок 1)

Настраиваем систему контроля версий git с использованием сервера репозиторий <https://github.com/>. Сделаем предварительную конфигурацию, указав имя и email владельца репозитория (dvkasjyanov, kasyanoff.daniil@yandex.ru) (Рисунок 2).

```
[dvkasjyanov@dvkasjyanov ~]$ git config --global user.name "dvkasjyanov"
[dvkasjyanov@dvkasjyanov ~]$ git config --global user.email "kasyanoff.daniil@yandex.ru"
[dvkasjyanov@dvkasjyanov ~]$ █
```

(Рисунок 2)

Для последующей идентификации пользователя на сервере репозиторий необходимо сгенерировать пару ключей (приватный и открытый) (Рисунок 3):

```
ssh-keygen -C "dvkasjyanov <kasyanoff.daniil@yandex.ru>"
```

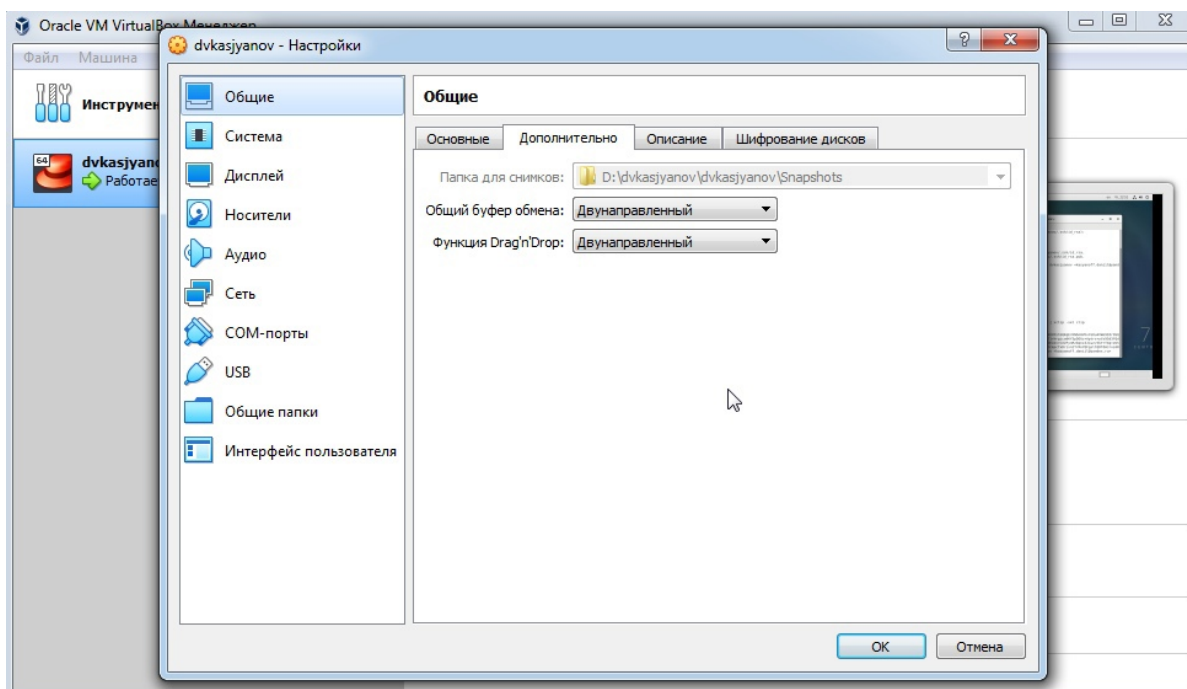
```

[dvkasjyanov@dvkasjyanov ~]$ git config --global user.name "dvkasjyanov"
[dvkasjyanov@dvkasjyanov ~]$ git config --global user.email "kasyanoff.daniil@yandex.ru"
[dvkasjyanov@dvkasjyanov ~]$ ssh-keygen -C "dvkasjyanov <kasyanoff.daniil@yandex.ru>"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/dvkasjyanov/.ssh/id_rsa):
/home/dvkasjyanov/.ssh/id_rsa already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/dvkasjyanov/.ssh/id_rsa.
Your public key has been saved in /home/dvkasjyanov/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:mRPbpouBMCD0WB8RQuvLrC49XHRaaD3qsRCKhWKZIVw dvkasjyanov <kasyanoff.daniil@yandex.ru>
The key's randomart image is:
+---[RSA 2048]-----+
|o+BEo.                |
|+*oo .                |
|o+= .o .              |
|*.o+.= *              |
|. +++= .S o           |
| o..=+ +              |
| o++o. .              |
|..+ o o .             |
|+. . . .              |
+----[SHA256]-----+
[dvkasjyanov@dvkasjyanov ~]$

```

(Рисунок 3)

Перехожу в «Настройки» ☒ «Общие» ☒ «Дополнительно» в виртуальной машине. В параметрах «Общий буфер обмена» и «Функция Drag'n'Drop» выбираю «Двунаправленный» (Рисунок 4). Это необходимо для того, чтобы ключ, сгенерированный в терминале Linux, можно было вставить в браузер в Windows.



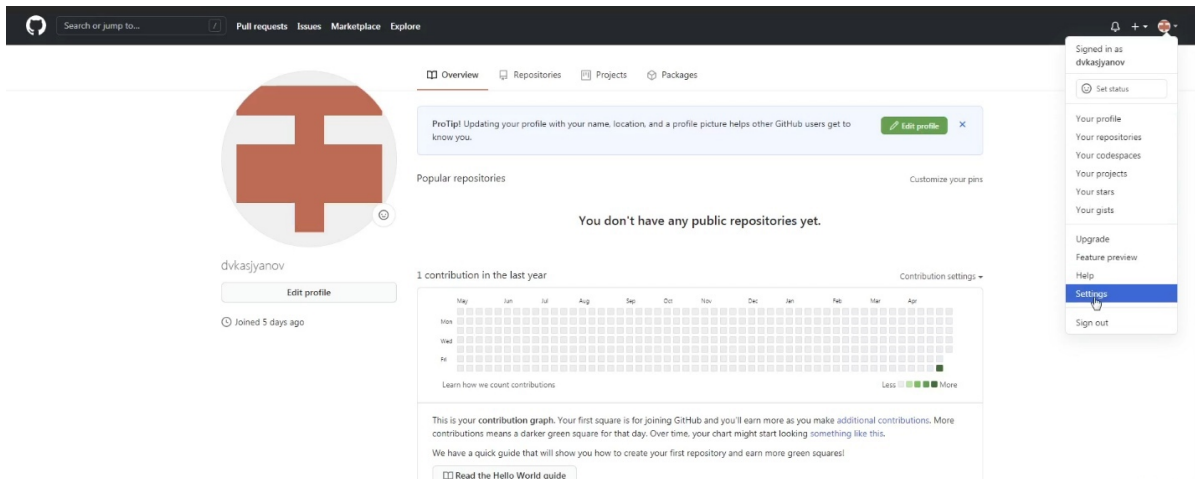
(Рисунок 4)

Используем команду `cat ~/.ssh/id_rsa.pub | xclip -sel clip` для копирования ключа в буфер обмена. В моем случае команда выдает ошибку, поэтому я использую команду `cat ~/.ssh/id_rsa.pub` и копирую сгенерированный ключ самостоятельно (Рисунок 5).

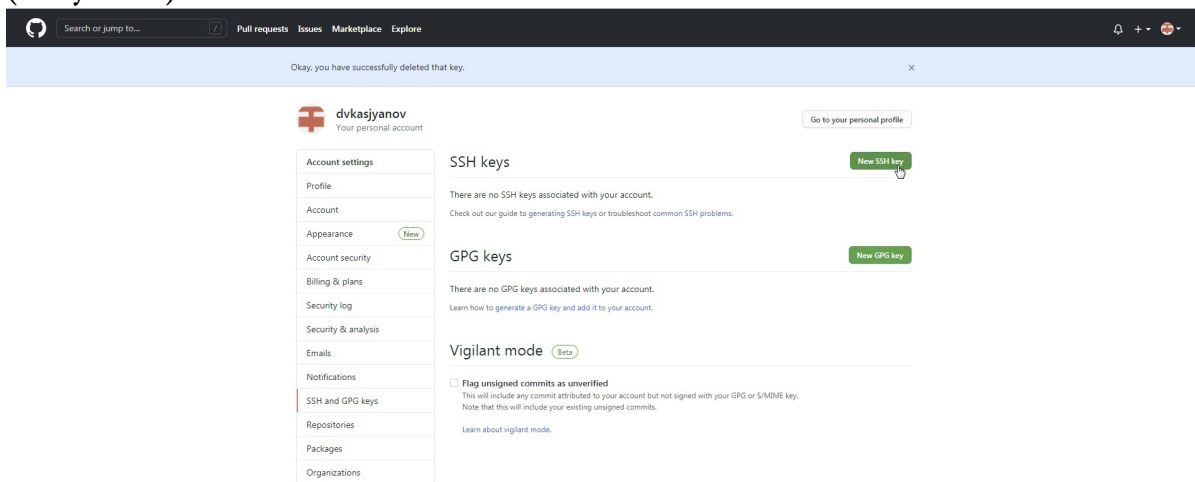
```
[dvkasjyanov@dvkasjyanov ~]$ cat ~/.ssh/id_rsa.pub | xclip -sel clip
bash: xclip: команда не найдена...
[dvkasjyanov@dvkasjyanov ~]$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDGpWjaLnUPTj40XPstGQKqErKNbn0VhvYe0u4VWo9I0r0SQ
qVmA3j1+JbuaBIUhs33yVL2Kfa0fxUZQFMBsmHFDhpeP41N/U471rHrgica0KY5pDEUz+Aptrs+yVz9Sd3fQd
ZN8QBk5HmgyyqvACRgZzrsAxnedPyz6CakaecFtG9pF5DUhohsZEiDv+CXZFynRzbqnzZJiwz/5kY+T4q+aVS
8IZ9pq706VJTMJoDVq4neHbPZBb/6VRTHHB1clvxlhKRfD4IdvJCmpcTxDrin2TrRnYQFgsltQDfdHJrxL6R
f5WBG9MX00UbmYbGBXlm/ZEpxtnIgv6AsF97tivJ dvkasjyanov <kasyanoff.daniil@yandex.ru>
```

(Рисунок 5)

Загрузим ключ на сайте <https://github.com/>. Переходим в меню «GitHub setting», выбираем в боковом меню «GitHub setting» (Рисунок 6), «SSH-ключи» и нажимаем кнопку «Добавить ключ» (Рисунок 7).

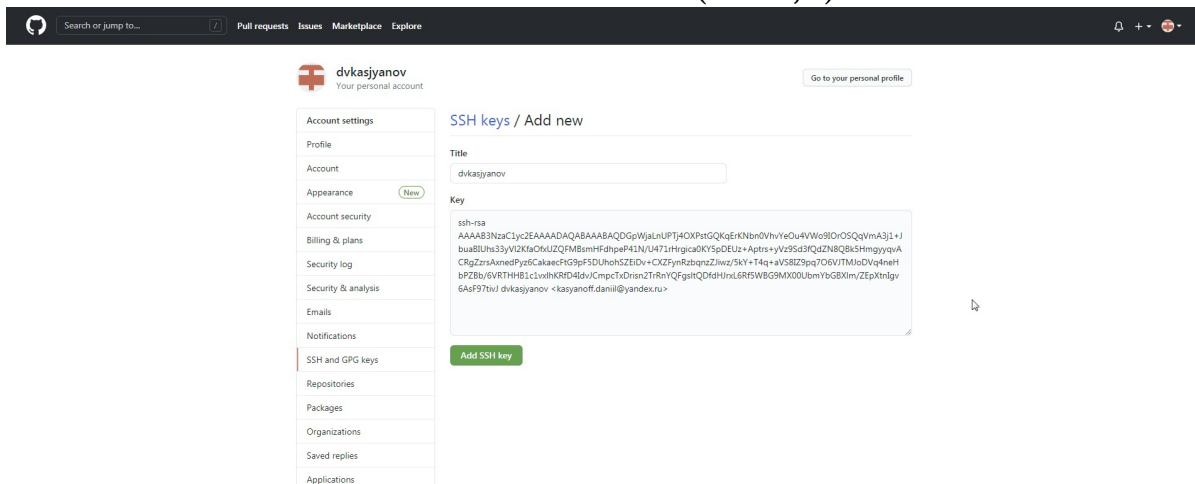


(Рисунок 6)

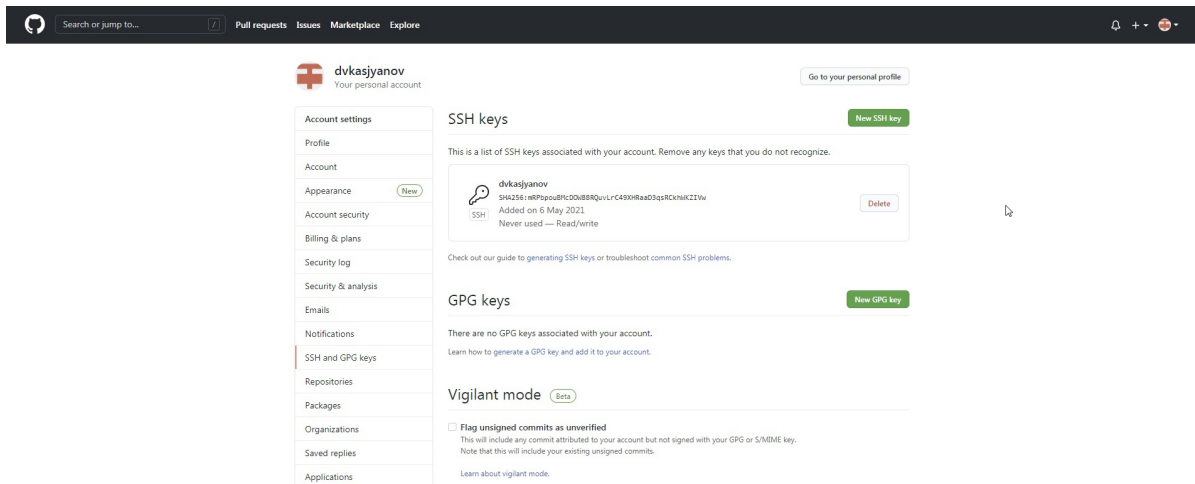


(Рисунок 7)

Вставляем ключ в появившееся на сайте поле (Рис. 8, 9).

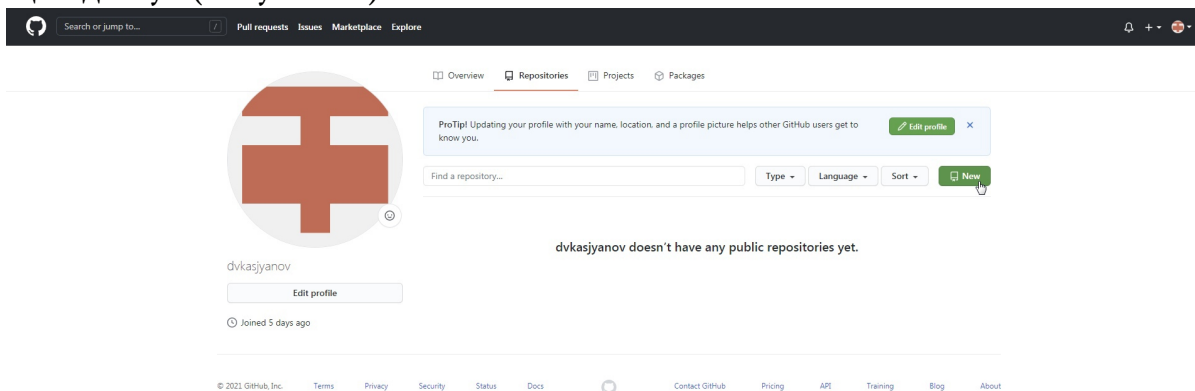


(Рисунок 8)

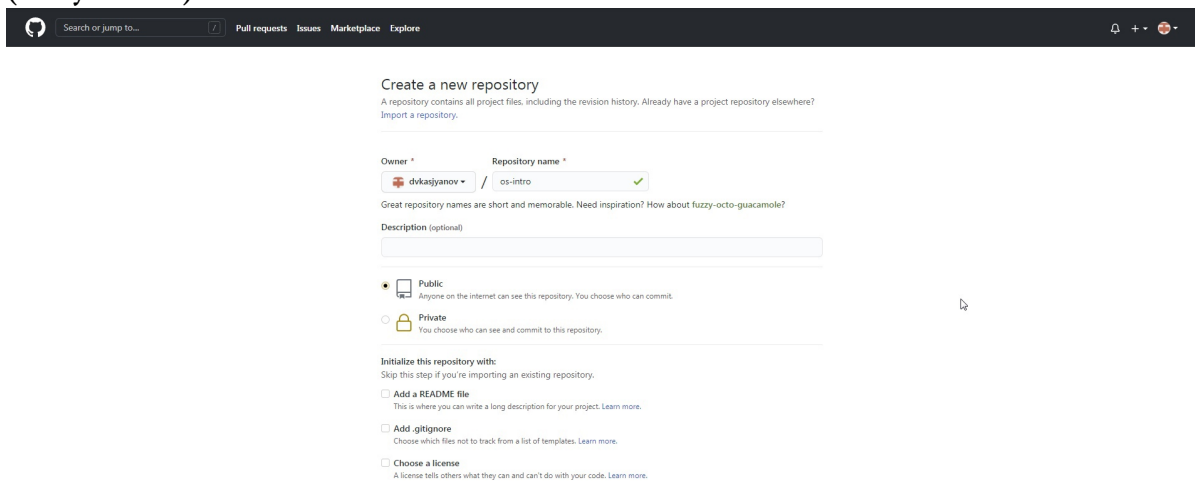


(Рисунок 9)

Создаём репозиторий на <https://github.com/>, выбрав в меню «Репозитории» **✕** «Создать репозиторий» (Рисунок 10). Назову репозиторий os-intro и открою общий доступ (Рисунок 11).



(Рисунок 10)



(Рисунок 11)

Создадим рабочий каталог `laboratory`, указав его расположение согласно соглашению об именовании (Рисунок 12).

```
[dvkasjyanov@dvkasjyanov ~]$ cd work
[dvkasjyanov@dvkasjyanov work]$ cd 2020-2021
[dvkasjyanov@dvkasjyanov 2020-2021]$ cd 05
[dvkasjyanov@dvkasjyanov 05]$ cd laboratory
[dvkasjyanov@dvkasjyanov laboratory]$
```

(Рисунок 12)

Инициализируем системы `git`:

```
git init
```

Создаём заготовку для файла `README.md`:

```
echo "# Лабораторные работы" >> README.md
```

```
git add README.md
```

Делаем первый коммит и выкладываем на `github` (Рисунок 13):

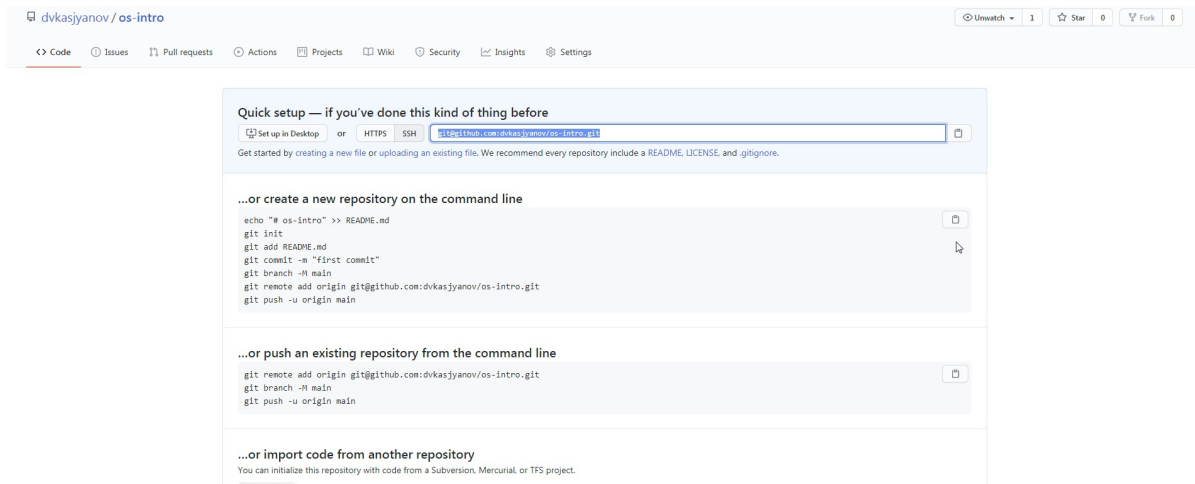
```
git commit -m "first commit"
```

```
[dvkasjyanov@dvkasjyanov laboratory]$ git init
Initialized empty Git repository in /home/dvkasjyanov/work/2020-2021/05/laboratory/.git/
[dvkasjyanov@dvkasjyanov laboratory]$ echo "# Лабораторные работы" >> README.md
[dvkasjyanov@dvkasjyanov laboratory]$ git add README.md
[dvkasjyanov@dvkasjyanov laboratory]$ git commit -m "first commit"
[master (root-commit) 740cfe0] first commit
1 file changed, 1 insertion(+)
 create mode 100644 README.md
[dvkasjyanov@dvkasjyanov laboratory]$ █
```

(Рисунок 13)

Копируем SSH ссылку на репозиторий с сайта `github` (Рисунок 14), выкладываем созданный репозиторий на `github`, используя ссылку в формате SSH (Рисунок 15):

```
git remote add origin git@github.com:<username>/sciproc-
intro.git
git push -u origin master
```



(Рисунок 14)

```
[dvkasjyanov@dvkasjyanov laboratory]$ git remote add origin git@github.com:dvkasjyanov/os-intro.git
[dvkasjyanov@dvkasjyanov laboratory]$ git push -u origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 250 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:dvkasjyanov/os-intro.git
* [new branch]      master -> master
Branch master set up to track remote branch master from origin.
[dvkasjyanov@dvkasjyanov laboratory]$
```

(Рисунок 15)

Проведём первичную конфигурацию. Добавим файл лицензии (Рисунок 16):

```
wget https://creativecommons.org/licenses/by/4.0/legalcode.txt -O LICENSE
```

```
[dvkasjyanov@dvkasjyanov laboratory]$ wget https://creativecommons.org/licenses/by/4.0/legalcode.txt -O LICENSE
--2021-05-06 22:59:26-- https://creativecommons.org/licenses/by/4.0/legalcode.txt
Распознаётся creativecommons.org (creativecommons.org)... 104.20.150.16, 172.67.34.140, 104.20.151.16, ...
Подключение к creativecommons.org (creativecommons.org)|104.20.150.16|:443... соединение установлено.
HTTP-запрос отправлен. Ожидание ответа... 200 OK
Длина: нет данных [text/plain]
Сохранение в: «LICENSE»

[ <=> ] 18 657 --.-K/s за 0,02s

2021-05-06 22:59:28 (1,04 MB/s) - «LICENSE» сохранён [18657]
[dvkasjyanov@dvkasjyanov laboratory]$
```

(Рисунок 16)

Добавим шаблон игнорируемых файлов. Просмотреть список имеющихся шаблонов можно с помощью команды:

```
curl -L -s https://www.gitignore.io/api/list
```

Скачаем шаблон для C (Рисунок 17):

```
curl -L -s https://www.gitignore.io/api/c >> .gitignore
```

```
[dvkasjyanov@dvkasjyanov laboratory]$ curl -L -s https://www.gitignore.io/api/c >> .gitignore
```

(Рисунок 17)

Добавим новые файлы:

```
git add .
```

Выполним коммит:

```
git commit -am 'my commit'
```

Отправим на github (Рисунок 18):

```
git push
```

```
[dvkasjyanov@dvkasjyanov laboratory]$ git add .
[dvkasjyanov@dvkasjyanov laboratory]$ git commit -am "my commit"
[master 1b03117] my commit
2 files changed, 455 insertions(+)
create mode 100644 .gitignore
create mode 100644 LICENSE
[dvkasjyanov@dvkasjyanov laboratory]$ git push
warning: push.default is unset; its implicit value is changing in
Git 2.0 from 'matching' to 'simple'. To squelch this message
and maintain the current behavior after the default changes, use:

    git config --global push.default matching

To squelch this message and adopt the new behavior now, use:

    git config --global push.default simple

See 'git help config' and search for 'push.default' for further information.
(the 'simple' mode was introduced in Git 1.7.11. Use the similar mode
'current' instead of 'simple' if you sometimes use older versions of Git)

Counting objects: 5, done.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 6.43 KiB | 0 bytes/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To git@github.com:dvkasjyanov/os-intro.git
740cfe0..1b03117 master -> master
[dvkasjyanov@dvkasjyanov laboratory]$ █
```

(Рисунок 18)

Инициализируем git-flow:

```
git flow init
```

Оставим префиксы master, develop, feature/, release/, hotfix/, support/ без изменений; префикс для ярлыков установим в v (Рисунок 19).

```
[dvkasjyanov@dvkasjyanov laboratory]$ git flow init

Which branch should be used for bringing forth production releases?
- master
Branch name for production releases: [master] master
Branch name for "next release" development: [develop] develop

How to name your supporting branch prefixes?
Feature branches? [feature/] feature/
Release branches? [release/] release/
Hotfix branches? [hotfix/] hotfix/
Support branches? [support/] support/
Version tag prefix? [] v
[dvkasjyanov@dvkasjyanov laboratory]$ █
```

(Рисунок 19)

Проверим, что мы на ветке develop: `git branch` (Рисунок 20).

```
[dvkasjyanov@dvkasjyanov laboratory]$ git branch
* develop
  master
[dvkasjyanov@dvkasjyanov laboratory]$ █
```

(Рисунок 20)

Создадим релиз с версией 1.0.0: `git flow release start 1.0.0` (Рисунок 21).

```
[dvkasjyanov@dvkasjyanov laboratory]$ git flow release start 1.0.0
Switched to a new branch 'release/1.0.0'

Summary of actions:
- A new branch 'release/1.0.0' was created, based on 'develop'
- You are now on branch 'release/1.0.0'

Follow-up actions:
- Bump the version number now!
- Start committing last-minute fixes in preparing your release
- When done, run:

    git flow release finish '1.0.0'

[dvkasjyanov@dvkasjyanov laboratory]$ █
```

(Рисунок 21)

Запишем версию: `echo "1.0.0" >> VERSION` (Рисунок 22).

```
[dvkasjyanov@dvkasjyanov laboratory]$ echo "1.0.0" >> VERSION
```

(Рисунок 22)

Добавим в индекс:

```
git add .
```

Сделаем commit (Рисунок 23):

```
git commit -am 'chore(main): add version'
```



```
[dvkasjyanov@dvkasjyanov laboratory]$ git add .
[dvkasjyanov@dvkasjyanov laboratory]$ git commit -am "chore(main): add version"
[release/1.0.0 69b8853] chore(main): add version
1 file changed, 1 insertion(+)
create mode 100644 VERSION
[dvkasjyanov@dvkasjyanov laboratory]$
```

(Рисунок 23)

Зальём релизную ветку в основную ветку (Рисунок 24):

`git flow release finish 1.0.0`

```
[dvkasjyanov@dvkasjyanov laboratory]$ git flow release finish 1.0.0
Branches 'master' and 'origin/master' have diverged.
And local branch 'master' is ahead of 'origin/master'.
Switched to branch 'develop'
Merge made by the 'recursive' strategy.
VERSION | 1 +
1 file changed, 1 insertion(+)
create mode 100644 VERSION
Deleted branch release/1.0.0 (was 69b8853).

Summary of actions:
- Latest objects have been fetched from 'origin'
- Release branch has been merged into 'master'
- The release was tagged 'v1.0.0'
- Release branch has been back-merged into 'develop'
- Release branch 'release/1.0.0' has been deleted

[dvkasjyanov@dvkasjyanov laboratory]$ █
```

(Рисунок 24)

В открывшихся окнах вводим произвольные сообщения (Рис. 25, 26).

```
Finish
#
# Write a tag message
# Lines starting with '#' will be ignored.
~
```

(Рисунок 25)

```
Merge branch 'release/1.0.0' into develop

it's necessary

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
```

(Рисунок 26)

Отправим данные на github (Рисунок 27):

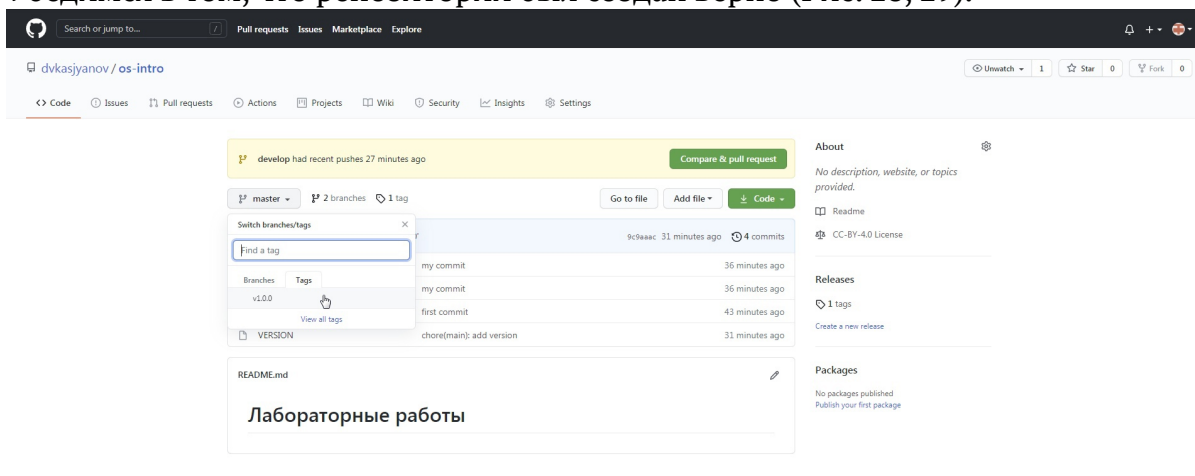
`git push --all`

`git push -tags`

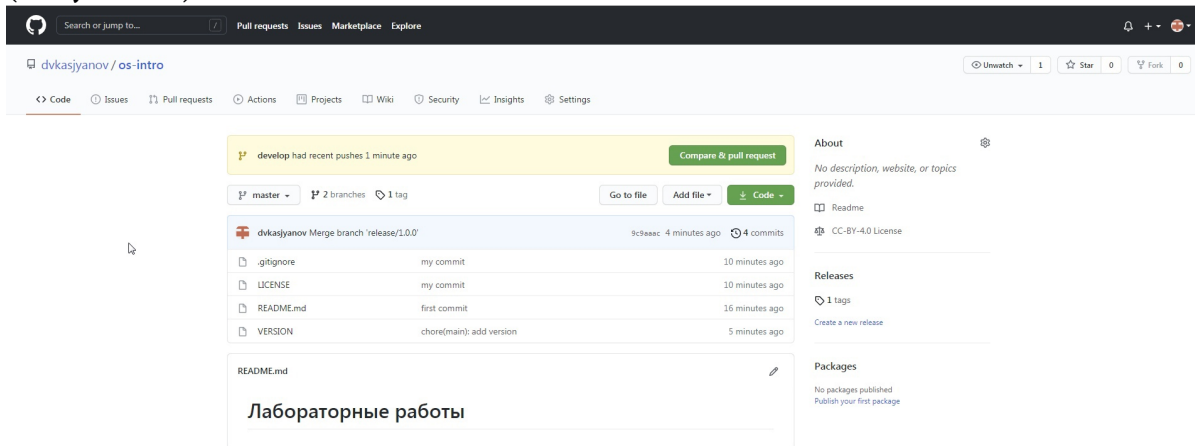

```
[dvkasjyanov@dvkasjyanov laboratory]$ git push --all
Counting objects: 6, done.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 436 bytes | 0 bytes/s, done.
Total 5 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 1 local object.
To git@github.com:dvkasjyanov/os-intro.git
    1b03117..9c9aaac  master -> master
    * [new branch]      develop -> develop
[dvkasjyanov@dvkasjyanov laboratory]$ git push --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 164 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:dvkasjyanov/os-intro.git
    * [new tag]         v1.0.0 -> v1.0.0
[dvkasjyanov@dvkasjyanov laboratory]$
```

(Рисунок 27)

Убедимся в том, что репозиторий был создан верно (Рис. 28, 29).



(Рисунок 28)



(Рисунок 29)

4 Контрольные вопросы

- 1) VCS (VersionControlSystem), система контроля версий – программное обеспечение, используемое для работы с различными версиями проекта; применяются при работе нескольких человек над одним проектом. Обычно основное дерево проекта хранится в локальном или удалённом репозитории, к которому настроен доступ для участников проекта. При внесении изменений в содержание проекта система контроля версий позволяет их фиксировать, совмещать изменения, произведённые разными участниками проекта, производить откат к любой более ранней версии проекта, если это требуется. Сервер может проводить дельта-компрессию, сохраняя не полные версии проекта, а изменения между последовательными версиями, уменьшая объем хранимых данных и упрощая взаимодействие между локальным и удаленным репозиторием. Системы контроля версий поддерживают возможность отслеживания и разрешения конфликтов, которые могут возникнуть при работе нескольких человек над одним файлом. Можно объединить (слить) изменения, сделанные разными участниками (автоматически или вручную), вручную выбрать нужную версию, отменить изменения вовсе или заблокировать файлы для изменения. В зависимости от настроек блокировка не позволяет другим пользователям получить рабочую копию или препятствует изменению рабочей копии файла средствами файловой системы ОС, обеспечивая таким образом, привилегированный доступ только одному пользователю, работающему с файлом. Системы контроля версий также могут обеспечивать дополнительные, более гибкие

функциональные возможности. Например, они могут поддерживать работу с несколькими версиями одного файла, сохраняя общую историю изменений до точки ветвления версий и собственные истории изменений каждой ветви. Кроме того, обычно доступна информация о том, кто из участников, когда и какие изменения вносил. Обычно такого рода информация хранится в журнале изменений, доступ к которому можно ограничить.

- 2) Хранилище (репозиторий) - место, где хранятся и поддерживаются какие-либо данные. Сохранение локальных изменений, синхронизация с репозиторием производится с помощью команды `commit` (в системе `git` для загрузки изменений на репозитории используется затем команда `push`). `Commit` так же некоторая точка сохранения версии. История – список всех изменений. Рабочая копия – копия проекта из репозитория, над которой проводит изменения (с которой работает) пользователь.
- 3) Централизованная VCS: один центральный репозиторий, с которым работчики взаимодействуют по сети или локально. Примеры: CVS, SVN (Subversion). Распределенные VCS (DVCS) не нуждаются в централизованном хранилище: вся история изменения документов хранится на каждом компьютере, в локальном хранилище, и при необходимости отдельные фрагменты истории локального хранилища синхронизируются с аналогичным хранилищем на другом компьютере. В некоторых таких системах локальное хранилище располагается непосредственно в каталогах рабочей копии. Примеры: Git, Mercurial, Bazaar.
- 4)
 - Создать новый репозиторий на локальном устройстве (если он не был создан);
 - Внести изменения в исходные файлы;
 - Выполнить индексацию необходимых файлов;

- Проверить внесенные изменения;
- Выполнить commit;
- Отправить локальный репозиторий на удаленный сервер, при необходимости.

5)

- Загрузить проект из общего удаленного хранилища;
- Необходимо создать свою рабочую ветку;
- Внести изменения внутри своей рабочей ветки;
- Выполнить индексацию необходимых файлов на своем локальном устройстве;
- Проверить внесенные изменения;
- Выполнить commit;
- Свою рабочую ветку отправить в общее хранилище;
- При необходимости внести изменения и отправить снова;
- После администратор объединит вашу ветку с master branch.

6) У Git две основных задачи: первая – хранить информацию о всех изменениях в проекте, а вторая – обеспечение удобства командной работы над кодом. Git запоминает не все изменения, а только те, которые указывает пользователь. Осуществляется это при помощи индекса репозитория (посредник между компьютером и удаленным репозиторием) и команд `git add`, `git rm`.

7) `git init` – создание (инициализация) основного дерева репозитория;

`git pull` - получение обновлений (изменений) текущего дерева из центрального репозитория;

`git push` - отправка всех произведённых изменений локального дерева в центральный репозиторий;

`git status` - просмотр списка изменённых файлов в текущей директории

`git diff` - просмотр текущих изменений;

добавление текущих изменений:

`git add .` - добавить все изменённые и/или созданные файлы и/или каталоги;

`git add имена_файлов` - добавить конкретные изменённые и/или созданные файлы и/или каталоги;

`git rm имена_файлов` - удалить файл и/или каталог из индекса репозитория (при этом файл и/или каталог остаётся в локальной директории);

сохранение добавленных изменений:

`git commit -am 'Описание коммита'` - сохранить все добавленные изменения и все изменённые файлы;

`git commit` - сохранить добавленные изменения с внесением комментария через встроенный редактор;

`git branch` – создание ветки;

`git checkout имя_ветки` - переключение на некоторую ветку;

`git checkout -b имя_ветки` - создание новой ветки, базирующейся на текущей (заменяет команды `git branch` и `git checkout имя_ветки`);

`git push origin имя_ветки` - отправка изменений конкретной ветки в центральный репозиторий;

`git merge --no-ff имя_ветки` - слияние ветки с текущим деревом;

удаление ветки:

`git branch -d имя_ветки` - удаление локальной уже слитой с основным деревом ветки;

`git branch -D имя_ветки` - принудительное удаление локальной ветки;

`git push origin :имя_ветки` - удаление ветки с центрального репозитория.

8) Примеры использования VCS при работе с локальными репозиториями:

Создание небольшого проекта на своем локальном устройстве, работа с файлами (например, каталог, содержащий документы, презентации, которые будут часто редактироваться), работа с фотографиями, видео и т.д.

Примеры использования VCS при работе с удаленными репозиториями:

Примеры могут быть те же, но теперь над ними работают несколько человек. Такая система позволяет следить за работой других пользователей.

9) Ветвление («ветка», branch) – один из параллельных участков истории в одном хранилище, исходящих из одной версии (точки ветвления).

- Обычно есть главная ветка (master), или ствол (trunk).
- Между ветками, то есть их концами, возможно слияние.

Ветки используются для разработки одной части функционала изолированно от других. Каждая ветка представляет собой отдельную копию кода проекта. Ветки позволяют одновременно работать над разными версиями проекта. Каждый репозиторий по умолчанию имеет ветку master. Всякий раз, когда требуется разработка нового функционала, не внося при этом изменений в основную рабочую версию, можно создавать новую ветку, на основе рабочей, и вести разработку в ней – новой копии кода проекта. Когда функционал доделан и протестирован, можно сделать merge – слить отдельную ветку обратно с основной. При слиянии этой временной ветки в основную, все её коммиты разом перенесутся из одной в другую. Ветвление позволяет обеспечить процесс, при котором всегда в наличии будет рабочая версия проекта, в которой не будет частично завершённого функционала находящегося в активной разработке или же протестированных фич.

- 10) Во время работы над проектом так или иначе могут создаваться файлы, которые не требуется добавлять в последствии в репозиторий. Например, временные файлы, создаваемые редакторами, или объектные файлы, создаваемые компиляторами. Можно прописать шаблоны игнорируемых при добавлении в репозиторий типов файлов в файл `.gitignore` с помощью сервисов. Для этого сначала нужно получить список имеющихся шаблонов: `curl -L -s https://www.gitignore.io/api/list`. Затем скачать шаблон, например, для C и C++: `curl -L -s https://www.gitignore.io/api/c`
>> `.gitignore` `curl -L -s https://www.gitignore.io/api/c++`
>> `.gitignore`.

5 Выводы

Я изучил идеологию и применение средств контроля версий.