



Дисципліна «Операційні системи»

Лабораторна робота №3

Тема: «Проста обробка текстових даних засобами оболонки Unix-подібних ОС інтерпретатора команд ОС»

Викладач: Олександр А. Блажко,
доцент кафедри ІС Одеської політехніки, blazhko@ieee.org

Мета роботи: придбання навичок роботи з оболонкою *Bash Unix*-подібних ОС інтерфейсу інтерпретатора команд ОС та простими командами обробки текстових даних.

1 Теоретичні відомості

1.1 Коротенька історія розвитку файлових систем ОС

Відомо, що назви більшості перших ОС для персональних комп'ютерів, наприклад, *Apple DOS*, *MS-DOS*, містили слово *Disk OS* – дискова, вказуючи на важливість для будь-якої ОС мати ефективну підсистему довгострокового зберігання даних.

Після впровадження у великі ОС механізму абстрагування периферійних пристроїв у вигляді файлів різного призначення, підсистеми зберігання даних стали називатися файловими підсистемами, в якій файл – це набір даних, до якого можна звертатися за іменем, а керування файлами забезпечувалося через виконання над ними базових команд роботи з будь-яким об'єктом:

- створення файлу;
- перегляд змісту файлу;
- зміна змісту файлу;
- видалення файлу.

Наведемо перелік файлових систем за хронологією їх появи:

- *FS (File System)* – перша файлова система для перших ОС *Unix*;
- *UFS (Unix File System)* – файлова система, створена для ОС сімейства *Unix BSD*;
- *FAT8 (File Allocation Table)* – файлова система для *DOS*, яка містить таблицю розміщення файлів у вигляді ланцюжка кластерів (логічне розбиття диску у вигляді 1, 2, 4, 8, 16 та більше 512-байтних секторів) різного розміру, наприклад:

- *FAT8* містила $2^8 = 256$ елементів таблиці, коли *max* розмір диска (файлу) = $256 \times 8\text{Кб} = 2\text{ Мб}$ для 16-секторних кластерів (8Кб);
- *FAT12* – файлова система для ОС *MS DOS 1.0-2.0* *max* розміром таблиці 2^{12} елементів;

- *FAT16* – файлова система для ОС *MS DOS 3.0* та *Windows 1.0-3.0* *max* розміром таблиці 2^{16} елементів;
- *FAT32* – файлова система для ОС *Windows 95* та *max* розміром таблиці 2^{32} елементів;
- *HPFS (High Performance File System)* – файлова система для ОС *IBM OS/2*, як альтернативи ОС *Windows 2.0-3.0*;
- *HFS (Hierarchical File System)* - файлова система ОС *Mac OS* з *max* розміром таблиці 2^{16} елементів; *HFS+* - розвиток файлової системи *HFS* з *max* розміром таблиці 2^{32} елементів та підтримкою файлів з кодуванням *Unicode* для імен файлів і каталогів;
- *NTFS (New Technology File System)* - файлова система для ОС *Windows NT*, створена на основі *HPFS*;
- *Ext (Extended File System)* – файлова система для ОС *Linux*, створена на основі файлової системи ОС *Minix* для розширення обмеження розміру файлу до 2Гб та довжини імені файлу до 256 символів;
- *Ext2* – розвиток файлової системи *Ext* ОС *Linux* з підвищеною швидкістю роботи та керування доступу до файлів у вигляді *ACL*-списків;
- *Ext3* – варіант файлової системи *Ext2* з журналюванням з передбаченим записом деяких даних, який дозволяє відновити файлову систему при збоях в роботі комп'ютера, та *max* розміром файлу до 2^{40} байтів, розміром диску до 2^{45} байт;
- *Ext4* – розвиток файлової системи *Ext3* з механізмом запису файлів в безперервні ділянки блоків (екстенти) для зменшення фрагментації (розміщення кластерів зі змістом одного файлу в різних частинах диску) та підвищення продуктивності *max* розміром файлу до 2^{44} байтів та розміром диску до 2^{60} байт.

Більшість сучасних файлових систем мають ієрархічну структуру у вигляді дерева:

- дерево починається з кореневого файлу – (*root* - вершини дерева), який для різних ОС може мати різні умовні позначення:
 - для файлових систем *FAT* та *NTFS* назва може бути *C:*, *D:*
 - для файлових систем *NFS*, *Ext* назва – це символ слеш /
- якщо файл розглядається як сховище інших файлів, тоді він стає каталогом;
- файли повинні мати унікальну назву в межах одного каталогу.

1.2 Навігація по файловій системі у *Bash*-оболонці командного рядку

1.2.1 Огляд файлової системи через *Bash*-оболонку командного рядку

Колись перші користувачі ОС *Unix* та їх послідовники з *Unix*-подібних ОС як гравці-дослідники якоїсь квест-гри намагалися відповідати на вищезгадані запитання, виконуючи лише команди у терміналі оболонки командного рядку ОС (*Shell*), наприклад, у *Thompson*-оболонці для ОС *Unix* першої редакції (1971 рік), у *Bourne*-оболонці для ОС *Unix* сьомої редакції (1976 рік) або у її нащадку – *Bash*-оболонці (*Bourne again shell*).

Програма *Bash* є частиною системи контролю версій *Git*, яку було використано у лабораторній роботі №2, тому в подальшому будемо розглядати саме цю програму.

Програма *Bash* як програма-інтерпретатор створювалася для обробки команд, які користувач вводить у командного рядку. Але слід зазначити, що більшість команд, які користувач може вводити, не є частиною самої програми, бо це програми командного рядку, які є зовнішніми по відношенню до *Bash*. Такі програми також називають утилітами (англ. *Utility, Utility program*). Для того, щоб дізнатися, що користувач використовує, *Bash*-команду, яка є вбудованою у *Bash*, чи утиліту, можна використати спеціальну *Bash*-команду:

type команда

Якщо команда вкаже на текст «*is a shell builtin*», тоді це *Bash*-команда. Якщо команда вкаже на місце розташування файлу, тоді це утиліта.

Приклади виконання *type* наведено на рисунку 1 на прикладі використання програми *Git Bash*.

<pre>blazhko@ws-18170 MINGW64 ~ \$ type pwd pwd is a shell builtin blazhko@ws-18170 MINGW64 ~ \$ type whoami whoami is /usr/bin/whoami blazhko@ws-18170 MINGW64 ~ \$ type date date is /usr/bin/date blazhko@ws-18170 MINGW64 ~ \$ type echo echo is a shell builtin</pre>	<pre>blazhko@ws-18170 MINGW64 ~ \$ type less less is /usr/bin/less blazhko@ws-18170 MINGW64 ~ \$ type cd cd is a shell builtin blazhko@ws-18170 MINGW64 ~ \$ type rm rm is hashed (/usr/bin/rm) blazhko@ws-18170 MINGW64 ~ \$ type type type is a shell builtin</pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Рис. 1 – Приклади виконання команди *type* для

Але в подальшому не будемо враховувати таку різницю і всі варіанти будемо називати командами *Bash*-оболонки.

Для недосвідчених користувачів оболонки командного рядку використання команд командного рядку може перетворитися на квест (англ. *quest*) – один з відомих жанрів

відеогор , який складається із вирішення гравцем поставлених завдань шляхом їх обдумування, уважного пошуку підказок і схованих деталей. Традиційно, коли користувач, як гравець-дослідник, вперше опиняється у якомусь невідомому місці квест-гри, найчастіше він хоче відповісти на такі запитання:

- 1) В якому місці я знаходжусь?
- 2) Ким я є?
- 3) Коли все це зі мною відбувається?
- 4) Де можна знайти інструменти та знаряддя?
- 5) Що знаходиться поряд зі мною?
- 6) Куди піти далі?
- 7) Як не заблукати?

В таблиці 1 наведено приклади команд, які є навігаційними для надання відповіді на наведені раніше квест-питання.

Таблиця 1 – Приклади команд *Bash*-оболонки як засоби гравця квест-гри

Питання квест-гри	Команда <i>Bash</i> -оболонки	Призначення команди <i>Bash</i> -оболонки
В якому місці я знаходжусь?	<i>pwd</i>	– (<i>print working directory</i> – надрукувати робочий каталог) – переглянути повний шлях від кореневого каталогу до поточного робочого каталогу
Ким я є?	<i>whoami</i>	переглянути ім'я користувача, який запустив оболонку
Коли все це зі мною відбувається?	<i>date</i>	переглянути поточну дату із визначенням окремих показників часу, наприклад, команда: <i>date +%e день %m місяця %G року</i> – виведе день (%e), місяць (%m) та рік (%G), але більше форматів можна отримати через виклик <i>date –help</i>
Де можна знайти інструменти та знаряддя?	<i>echo</i> <i>\$змінна_середовища</i>	переглянути значення змінної оточуючого середовища ОС, першим прикладом, якої є змінна <i>\$PATH</i> із переліком каталогів, необхідних для швидкого пошуку файлів, які можуть виконуватися в командному рядку без вказування повного шляху до файлу, при цьому каталоги розділяються символом двокрапка ':'

Таблиця 1 – продовження

Питання квест-гри	Команда <i>Bash</i> -оболонки	Призначення команди <i>Bash</i> -оболонки
Що знаходиться поряд зі мною?	<i>ls каталог</i>	(<i>list</i> – список) переглянути вміст каталогу за вказаною назвою або поточного каталогу, якщо назва не вказана
	<i>less файл</i>	переглянути вміст файлу (для завершення перегляду файлу використовується клавіша <i>q</i>).
Куди піти далі?	<i>cd каталог</i>	– (<i>change directory</i> – змінити каталог) перейти до іншого каталогу або домашнього каталогу користувача, якщо назва каталогу не вказана: – для переходу до каталогу на рівень вище (батьківський каталог) у назві каталогу вказуються символи дві крапки <i>..</i> , наприклад, <i>cd ..</i>
Як не заблукати?	<i>pushd каталог</i>	перейти до іншого каталогу за вказаною назвою з можливістю швидкого повернення до нього
	<i>popd</i>	повернутися до каталогу, з якого виконано переміщення після команди <i>pushd</i>

Більшість команд *Bash*-оболонки ОС пропонують розширений функціонал через так звані **прапори або ключі** – аргументи, що управляють роботою команди або вказують додаткові значення та позначаються через дефіс. Тире потрібно, щоб уникнути двозначності.

В таблиці 2 наведено приклади таких ключів для команди *ls*

Таблиця 2 – Параметри команди *ls*

Ключ	Опис ключа
<i>-a</i>	відображаються файли, назви яких починаються із крапки, - приховані файли
<i>-l</i>	відображається список файлів у псевдо табличному форматі із стовпчиками, в яких крім назви файлу вказуються, наприклад, розмір та дата його створення
<i>-R</i>	відображається вміст підкаталогів
<i>-S</i>	файли впорядковуються у відповідності з розміром

Ключі можна використовувати сумісно, наприклад: *ls -la*

1.2.2 Редагування каталогів та файлів файлової системи

Традиційно, коли користувач, як гравець-дослідник, вже отримав своє перше завдання у квест-грі, він виконує активні дії за умови наявності відповідей на три запитання:

- 1) Як щось створити, наприклад, місце для розташування знаряддя або скарбів?
- 2) Як щось перенести в інше місце?
- 3) Як щось видалити?

Якщо для активних дій квест-гри гравцям необхідно лише виконувати команди *Bash*-оболонки, тоді гравці можуть використати команди, представлені у таблиці 3.

Таблиця 3 – Приклади команд *Bash*-оболонки як засоби виконання гравцем елементарних дій із сутностями у квест-грі

Активна дія квест-гри		Команда <i>Bash</i> -оболонки	Призначення команди <i>Bash</i> -оболонки
Як щось створити?	створити місце для зберігання сутностей	<i>mkdir</i> <i>каталог</i>	створити каталог з назвою <i>каталог</i>
	створити сутність	<i>touch</i> <i>файл</i>	створити порожній файл з назвою <i>файл</i>
	змінити сутність	<i>ex</i> , <i>vi</i> , <i>nano</i> ,	розпочати роботу з редагування текстового файлу у редакторі файлів для ОС Linux: <i>ex</i> – подробиці https://uk.wikipedia.org/wiki/Ex_(Unix) <i>vi</i> – подробиці https://uk.wikipedia.org/wiki/Vi ; <i>nano</i> – подробиці https://uk.wikipedia.org/wiki/Nano
Як перенести сутність у інше місце зберігання?		<i>mv</i> <i>файл</i> <i>каталог</i>	перенести (<i>move</i>) файл з назвою <i>файл</i> до каталогу з назвою <i>каталог</i> , за необхідністю змінивши назву файлу
Як скопіювати сутність у інше місце зберігання?		<i>cp</i> <i>файл</i> <i>каталог</i>	Скопіювати (<i>copy</i>) файл з назвою <i>файл</i> до каталогу з назвою <i>каталог</i> , за необхідністю змінивши назву файлу
Як видалити сутність або місце зберігання сутностей?		<i>rm</i> <i>файл</i>	видалити файл з назвою <i>файл</i>
		<i>rmdir</i> <i>каталог</i>	видалити порожній каталог з назвою <i>каталог</i>
		<i>rmdir -f</i> <i>каталог</i>	видалити каталог, який містить файли без необхідності підтверджувати видалення файлу
		<i>rmdir -r</i> <i>каталог</i>	видалити каталог, який містить файли, рекурсивно видаляючи вкладені каталоги

1.2.3 Налаштування роботи *Bash*-оболонки

В процесі роботи з оболонкою користувач часто вимушений повторювати виконання якихось команд з різноманітним переліком прапорців. Для того, щоб не запам'ятовувати назви команд та перелік прапорців користувач може створити їх короткі назви, які користувачу легко запам'ятати та повторно виконувати. Для цього можна використати команду *alias* (псевдонім).

Команда *alias скорочена_форма_команди="повна форма команди"* – команда створення псевдонімів для скорочення команд та їх послідовностей (з використанням одинарних або подвійних лапок).

Наприклад, для перегляду списку файлів у псевдо табличному вигляді лише через псевдонім *ll* необхідно виконати наступну команду:

```
alias ll="ls -l"
```

Але вказана команда діє лише під час поточної роботи з оболонкою, коли після закриття оболонки всі дії будуть загублені. Щоб команди не треба було повторно створювати, необхідно їх додати до файлів конфігурації оболонки.

В *Unix*-подібних ОС, наприклад ОС *Linux*, результати налаштування оболонки *Bash* (конфігурація оболонки) можуть зберігатися у прихованих файлах для всієї ОС або у прихованих файлах домашніх каталогів користувачів, назви яких можуть різнитися, наприклад: *.bashrc*, *.bash_profile*, *.profile*.

Для *GitBash*-оболонки використовується файл *.bash_profile*, який повинен бути розміщений у домашньому каталозі користувача, тобто у початковому каталозі користувача, в який користувач потрапляє після старту оболонки.

1.3 Перенаправлення потоків даних

Відомо, що з розвитком ОС з'явилося «*Абстрагування від апаратних компонент*» через драйвери (англ. *Driver*) – програми, за допомогою яких ОС отримує доступ до різних раніше невідомих їй пристроїв:

- пристрій введення даних – *INPUT*;
- пристрій виведення даних – *OUTPUT*.

ОС керує «віртуальним пристроєм», який розуміє стандартний набір команд, а драйвер переводить команди ОС в команди, що розуміє пристрій.

В ОС *UNIX* існують такі стандартні потоки, які пов'язані зі спеціальними файлами:

- стандартний *INPUT*-потік з назвою *stdin* для файлу з файловим дескриптором = 0
- стандартний *OUTPUT*-потік – *stdout* для файлу з файловим дескриптором = 1
- стандартний потік помилок – *stderr* для файлу з файловим дескриптором = 2

ОС дозволяє користувачам та їх програмам керувати процесами перенаправлення стандартних потоків не на стандартні файли, а на будь-які інші файли.

Перенаправлення – це можливість командної оболонки перенаправляти стандартні потоки не на стандартні файли, а на будь-які інші.

Перенаправлення зазвичай здійснюється вставкою між командою та файлами спеціального символу > (більше) або символу < (менше).

У наступному прикладі виконується команда *команда1*, розміщуючи результат її роботи не у стандартний *stdout*-потік, наприклад, на екран, а у файл *файл1*, який автоматично створюється в поточному каталозі:

```
команда1 > файл1
```

При виконанні попередньої команди файл, який вже існував, буде видалено.

Використання пари символів >> дозволяє додати нові дані в кінець вже існуючого файлу:

```
команда1 >> файл1
```

У наступному прикладі виконується команда *команда1*, використовуючи в якості джерела отримання даних не стандартний *stdin*-потік, наприклад, клавіатуру, а зміст файлу *файл1*:

```
команда1 < файл1
```

У наступному прикладі поєднуються два попередні варіанти: виконується команда *команда1* отримання даних з файлу *файл1* і виведення даних у *файл2*:

```
команда1 < файл1 > файл2
```

У багатьох командних інтерпретаторах *UNIX*-подібних ОС, попередні дві дії можна вдосконалити, вказавши номер (значення файлового дескриптору), пов'язаний зі стандартним потоком, безпосередньо перед символом перенаправлення: 0 – *stdin*, 1 – *stdout*, 2 – *stderr*.

У наступному прикладі виконується команда видалення неіснуючого файлу *fl.txt* з перенаправленням стандартного потоку помилок не на екран через стандартний потік *stderr*, а у файл *error.txt*:

```
rm fl.txt 2> error.txt
```

Деякі оболонки допускають синтаксис вбудованих документів, що дозволяє направляти вхідний потік з самого файлу програми, наприклад, на стандартний *stdout*-потік, використовуючи команду *cat*:

```
cat << 'EOF'
```

Тут міститься довільний текст,

в тому числі – що включає в себе спеціальні символи

EOF

Або в файл, який буде автоматично створено:

```
cat << 'EOF' > файл
```

Тут міститься довільний текст,

в тому числі – що включає в себе спеціальні символи

EOF

Завершальна сигнатура закінчення вбудованого документу *EOF* (можна використовувати довільне значення, але часто використовується саме *EOF – End Of File*) повинна починатися з початку рядка.

Гарним прикладом перенаправлення потоків є використання команди *echo*, яка виводить рядок тексту на комп'ютерний термінал. Але перенаправивши результат не в *stdout*, а в будь-який файл, можна створити цей файл, наприклад, так буде створено файл *file.txt*, якому буде один рядок з символом *l*:

```
echo 'l' > file.txt
```

В подальшому можна додавати нові рядки до файлу через декілька команд:

```
echo '2' >> file.txt
```

```
echo '3' >> file.txt
```

1.4 Команди оболонки командного рядку з обробки тексту

1.4.1 Доступ до різних рядків текстового файлу

Іноді розмір текстового файлу, який виводиться на екран, значно більше кількості рядків на екрані, які користувач бачить одночасно. Щоб простити процес швидкого перегляду, можна переглядати лише частину перших або останніх рядків файлу, використовуючи команди *head* (голова) або *tail* (хвіст), відповідно:

head -n кількість_рядків – отримання з файлу перших рядків;

tail -n кількість_рядків – отримання з файлу останніх рядків;

Наприклад, для перегляду перших 10 рядків файлу *file.txt* виконується команда:

```
head -n 10 file.txt
```

1.4.2 Об'єднання декількох файлів в один файл

Команда *cat* (*concatenate*) забезпечує об'єднання (конкатенація) декількох файлів у стандартний вихідний потік. Також команду можна використовувати для перегляду змісту файлу, вказавши лише один файл.

Синтаксична конструкція використання:

```
cat файл1 файл2 ...
```

На рисунку 2 наведено приклади використання команди *cat*

```
Gamehub@DESKTOP-U9EG9KU
$ cat file1.txt
Б
А
В
Г
Е
Д
```

(а) – перегляд вмісту
файлу *file1.txt*

```
Gamehub@DESKTOP-U9EG9KU
$ cat file2.txt
2
1
4
5
3
6
```

(б) – перегляд вмісту
файлу *file2.txt*

```
Gamehub@DESKTOP-U9EG9KU MINGW64 ~
$ cat file1.txt file2.txt
Б
А
В
Г
Е
Д
2
1
4
5
3
6
```

(в) – об'єднання
файлів

Рис. 2 – Приклади використання команди *paste*

Команди *paste* також забезпечує об'єднання вмісту декількох файлів, але при цьому до вмісту кожного рядка першого файлу додається вміст відповідного за номером рядка другого файлу. Корисна опція команди *-d* – символ-роздільник між рядками файлів.

На рисунку 3 наведено приклади використання команди *paste*.

```
Gamehub@DESKTOP-U9EG9KU
$ cat file1.txt
Б
А
В
Г
Е
Д
```

(а)– вмісту файлу *file1.txt*

```
Gamehub@DESKTOP-U9EG9KU
$ cat file2.txt
2
1
4
5
3
6
```

(б)– вміст файлу *file2.txt*

```
Gamehub@DESKTOP-U9EG9KU MINGW64 ~
$ paste -d + file1.txt file2.txt
Б+2
А+1
В+4
Г+5
Е+3
Д+6
```

(в) – результат об'єднання файлів

Рис. 3 – Приклади використання команди *paste*

1.4.3 Сортювання елементів тексту

Для сортювання рядків у текстових файлах використовується команда *sort*.

Синтаксична конструкція використання:

sort [опції] файл -o вихідний_файл

Підтримуються такі опції:

- *r* – зворотнє сортювання;
- *k* – стовпчик1[, стовпчик2] починати сортювання зі стовпчика1, потім стовпчика2
- *n* – сортювання чисел;
- *t* символ – визначає символ-роздільник між колонками у рядку (за замовчуванням – прогалина);
- *u* – під час сортювання видаляються значення-дублікати.

На рисунку 4 наведено приклади сортювання даних в текстовому файлі *file1.txt*

```
Gamehub@DESKTOP-U9EG9KU
$ cat file1.txt
Б
А
В
Г
Е
Д
```

(а) – перегляд вмісту
файлу

```
Gamehub@DESKTOP-U9EG9KU
$ sort file1.txt
А
Б
В
Г
Д
Е
```

(б) – сортування за
зростанням коду символів

```
Gamehub@DESKTOP-U9EG9KU
$ sort -r file1.txt
Е
Д
Г
В
Б
А
```

(в) – сортування за
убуванням коду символів

Рис. 4 – Приклади сортування даних в текстовому файлі *file1.txt*

Приклад сортування рядків з урахуванням чисел з 5-ї колонки:

```
sort -nk 5 debts.txt
```

Приклад сортування у зворотному порядку з урахуванням значень 2-ї колонки, а потім 3-ї колонки:

```
sort -nk 2,3 debts.txt
```

Іноді необхідно лише провести аналіз тексту з урахуванням рядків-дублікатів. Хоча утиліта *sort* дозволяє видалити дублікати, для цього можна також використати команду *uniq* з опціями, яка додатково дозволяє не видалити, а навпаки – залишити рядки-дублі.

-*u* – видалення дублікатів рядків;

-*d* – видалення рядків, які не дублюються.

Але команда *uniq* буде працювати лише за умови наявності попередньо відсортованих даних, наприклад, командою *sort*, наприклад, з використанням за рахунок окремо створюваного файлу через перенаправлення потоку:

```
sort file1.txt > file_sort.txt
```

```
uniq -d < file_sort.txt
```

1.4.4 Просте перетворення символів рядку

Для простого перетворення символів текстового рядку можна використати команду *tr* (*translate* або *transliterate*), яка копіює стандартний вхідний потік в стандартний вихідний, підставляючи або видаляючи деякі символи.

Синтаксична конструкція використання:

```
tr [опції] набір1 [набір2]
```

В рядку *набір1* перераховуються символи, що підлягають заміні, а в рядку *набір2* у відповідному порядку перераховуються символи, які їх повинні замінити.

Підтримуються такі опції:

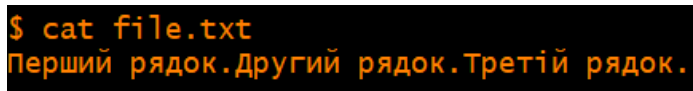
-*s* (*--squeeze-repeats*) – заміна входження символів, що поспіль повторюються, з рядка *набір1* на один символ, а якщо *набір2* відсутній, тоді множинні символи замінюються на поодинокі (стискаються);

-d (--delete) – видалення всіх входжень символів, вказаних в рядку *набір1*;
-c (--complement) – доповнення набору символів, що задається рядком *набір1*;
-t (--truncate-set1) – перед перетворенням виконується обрізка рядка *набір1* до довжини рядка *набір2*.

Наприклад, для автоматичної заміни символів точка на символи переводу рядка у файлі *file.txt*, приклад змісту якого наведено на рисунку 5, необхідно виконати команду:

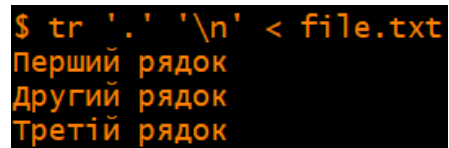
```
tr '.' '\n' < file.txt
```

Результат виконання команди наведено на рисунку 5.



```
$ cat file.txt
Перший рядок.Другий рядок.Третій рядок.
```

(a) – приклад вмісту файлу



```
$ tr '.' '\n' < file.txt
Перший рядок
Другий рядок
Третій рядок
```

(b) – результат роботи команди *tr*

Рис. 5 – Приклад перетворення символів командою *tr*

1.4.5 Простий статистичний аналіз тексту

wc (word count) - утиліта для підрахунку кількості рядків, слів чи байт у вказаних файлах, а також їх суму, якщо вказано більше одного файлу. Якщо файли не вказуються, тоді команда зчитує дані зі стандартного вводу.

Синтаксична конструкція використання:

```
wc [опції] файли...
```

Підтримуються такі опції:

-c, (--bytes) - виводить кількість байтів;

-l, (--lines) - виводить кількість рядків;

-L (--max-line-length) - виводить довжину найбільшого рядку;

-m - виводить кількість символів;

-w (--words) - виводить кількість слів у файлі;

Якщо опції не вказано, тоді статистика виводиться у наступному порядку:

кількість рядків, кількість слів, кількість байтів.

Вивести статистику (кількість рядків, кількість слів, кількість байтів) для файлів */etc/fstab* та */etc/passwd*, які знаходяться у файловій системі ОС *Linux*:

```
wc /etc/fstab /etc/passwd
```

Результат роботи команди:

```
[root@vpsj3IeQ ~]# wc /etc/fstab /etc/passwd
  11   54   370 /etc/fstab
 139  164 8313 /etc/passwd
 150  218 8683 total
```

1.5 Конвеєризація команд

Під час перенаправлення було розглянуто взаємозв'язки між командами та файлами в ОС. Але ОС також надає можливість встановлювати взаємозв'язки між різними командами.

Конвеєр – механізм міжкомандної (міжпроцесної) взаємодії, що забезпечує конвеєрну обробку даних, коли результати обробки, отримані в одній команді (процесі), передаються до іншої команди (процесу).

В *Unix*-подібних ОС конвеєрний обмін даними відбувається через так звані **неіменовані канали**, які забезпечують передачу даних так, що стандартний *stdout*-потік одної команди безпосередньо з'єднується зі стандартним *stdin*-потокм іншої команди. Програми, що використовуються у конвеєрі прийнято називати фільтрами, оскільки принцип їх роботи подібний до фільтрів-сит, через які «просіваються» дані.

В більшості *Unix*-подібних ОС процеси конвеєра запускаються одночасно та їхні стандартні потоки зв'язуються. Важливою особливістю реалізації конвеєрів в *Unix*-подібних ОС є застосування буферизації під час передачі даних. Завдяки буферизації, записування та зчитування даних у конвеєрі може відбуватись без звернення до зовнішніх пристроїв та із різною швидкістю без втрати даних.

Створення конвеєра в командному інтерпретаторі виконується командою:

```
command1 [arglist1] | command2 [arlist2]
```

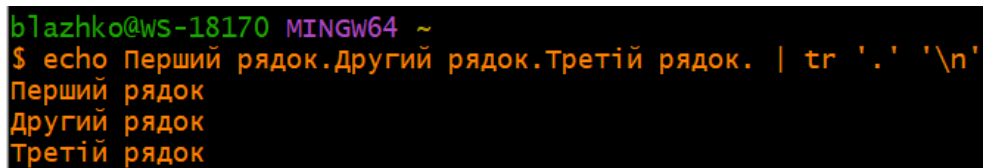
де: *command1*, *command2* - команди, між процесами яких має бути забезпечена обмін даними через канал; символ `|` - оператор створення неіменованого каналу.

Кількість команд у конвеєрі синтаксично не обмежена, але такі обмеження можуть встановлюватись ОС або командною оболонкою.

У наступному прикладі команда *echo* передає рядок тексту на стандартний *stdout*-потік, але через конвеєр цей потік автоматично зв'язується зі стандартним *stdin*-потокм команди *tr* для автоматичної заміни символів точка на символи переводу рядка:

```
echo Перший рядок.Другий рядок.Третій рядок. | tr '.' '\n'
```

Результат конвеєру команди представлено на рисунку 6.



```
blazhko@ws-18170 MINGW64 ~  
$ echo Перший рядок.Другий рядок.Третій рядок. | tr '.' '\n'  
Перший рядок  
Другий рядок  
Третій рядок
```

Рис. 6 – Приклад конвеєру команд

Приклад перекладу всіх символів з нижнього регістру у верхній регістр з використанням конвеєру для отримання вхідних даних від команди *echo*:

```
echo a black cat | tr 'a-z' 'A-Z'
```

```
[root@vpsj3IeQ ~]# echo a black cat | tr 'a-z' 'A-Z'
```

A BLACK CAT

echo дуже багато пробілів | tr -s ' '

```
[root@vps]3IeQ ~]# echo дуже        багато        пробілів | tr -s ' '
дуже багато пробілів
```

<pre>\$ echo \$PATH /c/Users/Gamehub/bin:/mingw64/bin:/usr/local/bin:/usr/bin:/bin:/mingw64/bin:/usr/bin:/c/Users/Gamehub/bin:/c/app/Gamehub/product/21c/dbhomeXE/bin:/c/oraclexe/app/oracle/product/11.2.0/server/bin:/d/oraclexe/app/oracle/product/11.2.0/server/bin:/c/windows/System32:/c/windows/c/windows/System32/wbem:/c/windows/System32/windowsPowerShell/v1.0:/c/Program Files (x86)/GtkSharp/2.12/bin:/c/windows/System32/OpenSSH:/c/Program Files (x86)/GtkSharp/2.12/bin:/c/Program Files/dotnet:/cmd:/c/Users/Gamehub/AppData/Local/Microsoft/WindowsApps:/c/Users/Gamehub/AppData/Local/Programs/Microsoft VS Code/bin:/usr/bin/vendor_perl:/usr/bin/core_perl</pre>	<pre>\$ echo \$PATH tr ':' '\n' /c/Users/Gamehub/bin /mingw64/bin /usr/local/bin /usr/bin /bin /mingw64/bin /usr/bin /c/Users/Gamehub/bin /c/app/Gamehub/product/21c/dbhomeXE/bin /c/oraclexe/app/oracle/product/11.2.0/server/bin /d/oraclexe/app/oracle/product/11.2.0/server/bin /c/windows/System32 /c/windows /c/windows/System32/wbem</pre>
<p>(a) Результат команди <i>echo \$PATH</i></p>	<p>(b) Результат виконання конвеєра команд</p>

коли відбувається заміна символу двокрапка на символ переходу на новий рядок

```
echo a black cat | tr -d a
```

```
[root@vps]# echo a black cat | tr -d a
blk ct
```

```
tr -s '\n' < file.txt > new file.txt
```

Результат роботи команди:

```
[root@vpsj3IeQ ~]# cat file.txt
```

Рядок 1

Рядок 2

Рядок 3

```
[root@vpsj3IeQ ~]# tr -s '\n' < file.txt
```

Рядок 1

Рядок 2

Рядок 3

Замінити всі символи крім символу / пропуск та перенос рядка `\n` на символ `a`:

```
echo "unix/linux blog" | tr -c ' /\n' 'a'
```

Результат роботи команди:

```
[root@vpsj3IeQ ~]# echo "unix/linux blog" | tr -c ' /\n' 'a'
aaaa/aaaaa aaaa
```

Припустимо, що є файл `/etc/fstab`. Для збереження всіх символів в діапазоні `a-zA-Z` та заміни всіх інших символів на символ переносу рядка `\n` необхідно виконати команду:

```
tr -cs a-zA-Z '\n' < /etc/fstab
```

Фрагмент результату роботи команди:

```
[root@vpsj3IeQ ~]# tr -cs a-zA-Z '\n' < /etc/fstab
etc
fstab
Created
by
anaconda
on
wed
```

Хоча утиліта `tr` є корисною, але вона може працювати тільки з поодинокими символами. Для більш складного зіставлення із шаблоном і роботи з рядками рекомендується використовувати утиліти `sed` або `awk`, про які мова піде в наступних лабораторних роботах.

Bash-оболонка має різні команди, які можуть виконувати схожі дії.

При виборі команд рекомендується враховувати наступне:

- обираємо просту команду, більш орієнтовану на виконання вказаного завдання;
- обираємо складну команду, якщо її опції дозволяють одночасно виконати декілька невеличких завдань зі складного завдання.

2 Завдання до виконання

2.1 Документування рішень завдань лабораторної роботи

На відміну від попередніх лабораторних робіт ця лабораторна робота пропонує створити звітність за рішеннями завдань у вигляді єдиного електронного документу з використанням вбудованих для веб-сервісу *GitHub* засобів підтримки *Markdown* мови розмітки документів.

Звітність буде створено у файлі *README.md* з урахуванням *Markdown*-форматування, який буде розташовано на веб-сервісі *GitHub*.

Якщо завдання виконуються в ОС *Windows*, на локальному комп'ютері треба запустити оболонку *GitBash*, а якщо завдання виконуються в інших ОС, наприклад, в ОС *MacOS* або в ОС *Linux*, запусти будь-яку оболонку, яка підтримує команди *Bash*-оболонки, або сумісну з нею оболонку.

2.1.1 Особливості підготовки до процесу документування рішень

Використовуючи команди *Bash*, виконати наступні завдання.

2.1.1.1 Виконати безпечне клонування *GitHub*-репозиторію, який був наданий вам викладачем, створивши на локальному комп'ютері *Git*-репозиторій.

Перейти до каталогу із *Git*-репозиторієм.

2.1.1.2 Створити нову *Git*-гілку з назвою «*Laboratory-work-3*».

Перейти до роботи зі створеною гілкою.

2.1.1.3 Створити каталог з назвою «*Laboratory-work-3*». Перейти до каталогу.

2.1.1.4 В каталозі «*Laboratory-work-3*» створити порожній файл *README.md*, використовуючи команду інтерпретатору командного рядку *Bash*.

2.1.1.5 Використовуючи текстові редактори, які пропонуються оболонкою *Git Bash*, наприклад, текстовий редактор *nano*, додати до файлу *README.md* рядок тексту із темою лабораторної роботи: «*Проста обробка текстових даних засобами оболонки Unix-подібних ОС інтерпретатора команд ОС*».

Для рядка визначити *Markdown*-форматування як заголовок 2-го рівня.

Для збереження змін та завершення роботи в редакторі *nano* можна вказати комбінацію клавіш *Ctrl+O* та *Ctrl+X*, відповідно.

2.1.1.6 Виконати операції з оновлення *GitHub*-репозиторію змінами *Git*-репозиторія через послідовність *Git*-команд *add*, *commit* із коментарем «*Changed by Local Git*» та *push*.

2.1.1.7 На веб-сервісі *GitHub* перейти до створеної гілки «*Laboratory-work-3*».

Перейти до каталогу «*Laboratory-work-3*» та розпочати процес редагування файлу *README.md*

Перейти до виконання завдань розділу 2.2.

2.1.2 Особливості проведення процесу документування рішень

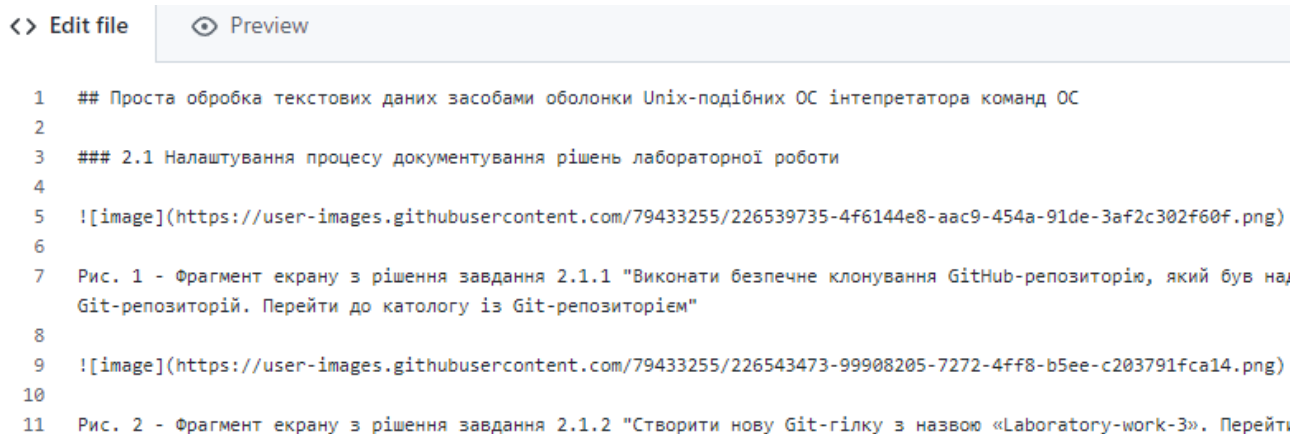
Після виконання наступних завдань лабораторної роботи планується додавати рішення завдань до файлу *README.md* у вигляді фрагментів знімків екранів рішень завдань та рядків-підписів із коментарями цих знімків.

Для кожного пункту виконаних завдань треба буде підготувати звітність, редагуючи файл *README.md*, який знаходиться у каталозі «*Laboratory-work-3*» гілки «*Laboratory-work-3*» у *GitHub*-репозиторії на веб-сервісі *GitHub*. На рисунку 5 наведено приклади створення звітності, які враховують, що під час виконання кожного пункту завдань необхідно:

- додати фрагмент знімку екрану, який розмістити у файлі через *Markdown*-форматування `![image](file)`, де *file* – повний шлях до файлу зі знімком, розташованому на веб-сервісі *GitHub*;

- додати текстовий рядок за шаблоном «*Рис. N – Фрагмент екрану з рішення завдання task*», де *N* – порядковий номер рисунку (1,2,3 ...), *task* – текст із номером та вмістом завдання.

Для швидкого розташування знімку можна скористатися класичними засобами *Copy/Paste* або *Drag-and-Drop*, які підтримуються веб-сервісом *GitHub*, тоді сервіс автоматично розмістить файл зі знімком у сховищі та надасть на нього посилення у файлі *README.md* за *Markdown*-форматуванням, як це показано на рисунку 8.



The image shows a screenshot of a GitHub repository's README.md file in edit mode. At the top, there are two tabs: 'Edit file' and 'Preview'. Below the tabs, the content of the README.md file is displayed in a code editor. The file contains a list of tasks, each followed by a screenshot. The tasks are numbered 1 through 11. The screenshots are linked to GitHub's user image content service. The text is as follows:

```
1  ## Проста обробка текстових даних засобами оболонки Unix-подібних ОС інтерпретатора команд ОС
2
3  ### 2.1 Налаштування процесу документування рішень лабораторної роботи
4
5  ![image](https://user-images.githubusercontent.com/79433255/226539735-4f6144e8-aac9-454a-91de-3af2c302f60f.png)
6
7  Рис. 1 - Фрагмент екрану з рішення завдання 2.1.1 "Виконати безпечне клонування GitHub-репозиторію, який був на
   Git-репозиторій. Перейти до каталогу із Git-репозиторієм"
8
9  ![image](https://user-images.githubusercontent.com/79433255/226543473-99908205-7272-4ff8-b5ee-c203791fca14.png)
10
11 Рис. 2 - Фрагмент екрану з рішення завдання 2.1.2 "Створити нову Git-гілку з назвою «Laboratory-work-3». Перейти
```

Рис. 8 – Фрагмет файлу *README.md* з описом знімків екранів рішення завдань

На рисунку 9 наведено фрагмент візуалізації вмісту файлу *README.md* зі знімками екранів рішення завдань.

Проста обробка текстових даних засобами оболонки Unix-подібних ОС інтерпретатора команд ОС

2.1 Налаштування процесу документування рішень лабораторної роботи

```
blazhko@ws-18170 MINGW64 ~  
$ git clone git@github.com:oleksandrblazhko/student_test.git  
Cloning into 'student_test'...  
remote: Enumerating objects: 24, done.  
remote: Counting objects: 100% (24/24), done.  
remote: Compressing objects: 100% (16/16), done.  
remote: Total 24 (delta 0), reused 0 (delta 0), pack-reused 0  
Receiving objects: 100% (24/24), 7.48 KiB | 1.87 MiB/s, done.  
  
blazhko@ws-18170 MINGW64 ~  
$ cd student_test/
```

Рис. 1 - Фрагмент екрану з рішення завдання 2.1.1 "Виконати безпечне клонування GitHub-репозиторію, який був наданий вам викладачем, створивши на локальному комп'ютері Git-репозиторій. Перейти до каталогу із Git-репозиторієм"

```
blazhko@ws-18170 MINGW64 ~/student_test (main)  
$ git branch Laboratory-work-3  
  
blazhko@ws-18170 MINGW64 ~/student_test (main)  
$ git checkout Laboratory-work-3  
Switched to branch 'Laboratory-work-3'  
  
blazhko@ws-18170 MINGW64 ~/student_test (Laboratory-work-3)
```

Рис. 2 - Фрагмент екрану з рішення завдання 2.1.2 "Створити нову Git-гілку з назвою «Laboratory-work-3». Перейти до роботи зі створеною гілкою."

Рис. 9 – Фрагмент файлу візуалізації вмісту файлу *README.md*
зі знімками екранів рішення завдань

2.2 Навігація по файловій системі через засоби оболонки *Git Bash* інтерпретатору командного рядку *Bash*

Документуючі рішення цього розділу, у файлі *README.md* необхідно вказати наступний текстовий рядок у вигляді заголовку 3-го рівня *Markdown*-форматування:

«1 Навігація по файловій системі через засоби оболонки *Git Bash* інтерпретатору командного рядку *Bash*». В подальшому за результатами рішень кожного пункту завдання до файлу *README.md* додати знімки екрану та коментарі з урахуванням рекомендацій пункту 2.1.2.

Знаходячись у кореневому каталозі *Git*-репозиторія, виконати наступні завдання.

2.2.1 Отримати перелік файлів поточного каталогу з урахуванням видимості прихованих файлів.

2.2.2 Перейти до прихованого каталогу *.git*, використовуючи команду *pushd* з метою швидкого повернення до попереднього каталогу у майбутньому.

2.2.3 Переглянути вміст файлу *config*, використовуючи редактор *nano*.

Для завершення перегляду вказати комбінацію клавіш *Ctrl+X*.

2.2.4 Отримати перелік файлів поточного каталогу з урахуванням наступних умов:

- відображення списку файлів у псевдо табличному форматі;
- впорядкування порядку слідування файлів за збуванням їх розміру.

2.2.5 Повернутися до каталогу, використовуючи команду швидкого повернення.

2.3 Налаштування псевдонімів команд оболонки *Bash*

Документуючі рішення цього розділу, у файлі *README.md* необхідно вказати наступний текстовий рядок у вигляді заголовку 3-го рівня *Markdown*-форматування:

«2 Налаштування псевдонімів команд оболонки *Bash*». В подальшому за результатами рішень кожного пункту завдання до файлу *README.md* додати знімки екрану та коментарі з урахуванням рекомендацій пункту 2.1.2.

Відомо, що однією з умов перетворення даних у інформацію є їх представлення мовою споживача, однією з яких є мова професіонального спілкування. Якщо спеціалістам різних галузей народного господарства, далеких від ІТ-галузі, надати можливість виконувати команди в оболонці *Bash*, є сумніви, що вони це зроблять без ускладнень. Якщо користувачу надати можливість виконувати команди, назви яких є синонімами назв знайомих їм процесів, тоді зростає ймовірність правильного виконання команд, якщо користувач не знайомий з командами інтерпретатора *Bash*. В таблиці 4 представлено напрями народного господарства, назви *Bash* команд та гіпотетичні (можливо, жартівливі) синоніми команд в процесах народного господарства.

Враховуючи вказані раніше припущення, необхідно виконати наступні завдання.

2.3.1 Виконати команду зі створення псевдоніму виклику команди, пов'язаною з *Bash* командою у відповідності з таблицею 4. Перевірити роботу псевдоніму команди.

2.3.2 Виконати команду зі створення псевдоніму виклику команди, яка буде надавати поточну дату лише із поточним днем, місяцем та роком. Назва псевдоніму визначається за шаблоном: «дата_» + «дія», де «дія» - значення синоніму команди з таблиці 4, наприклад, «дата_зібрати». При описі псевдоніму рекомендується використовувати подвійні лапки.

Перевірити роботу псевдоніму команди.

2.3.3 Перейти до домашнього каталогу вашого користувача. Використовуючи текстовий редактор, наприклад, *nano*, розпочати редагування файлу *.bash_profile* та додати у файл два рядки зі створеними раніше псевдонімами виклику команд. Для збереження змін та завершення роботи в редакторі *nano* вказати комбінацію клавіш *Ctrl+O* та *Ctrl+X*, відповідно.

2.3.4 Завершити роботу з *GitBash*-оболонкою через команду *exit*

2.3.5 Повторно запустити *GitBash*-оболонку та перевірити роботу одного зі створених псевдонімів команд, щоб підтвердити їх автоматичну реєстрацію через файл *.bash_profile*

2.3.6 Скопіювати файл *.bash_profile* до каталогу «*Laboratory-work-3*» *Git*-репозиторію

Таблиця 4 – Варіанти завдання для створення псевдонімів команд

№	Галузь народного господарства	<i>Bash</i> команди	Синонім команди у процесах народного господарства
1.	Будівництво	створити каталог	зібрати
2.	Будівництво	створити файл	встановити
3.	Будівництво	видалити файл	зняти
4.	Будівництво	переглянути файл	простукати
5.	Сільське господарство	створити каталог	засіяти
6.	Сільське господарство	створити файл	полити
7.	Сільське господарство	видалити файл	зібрати
8.	Сільське господарство	переглянути файл	прополоти
9.	Освіта	створити каталог	зарахувати
10.	Освіта	створити файл	перевести
11.	Освіта	видалити файл	відрахувати
12.	Освіта	переглянути файл	Опитати
13.	Банківська справа	створити каталог	відкрити
14.	Банківська справа	створити файл	Переоформити
15.	Банківська справа	видалити файл	закрити
16.	Банківська справа	переглянути файл	перерахувати
17.	Рибне господарство	створити каталог	зарибити
18.	Рибне господарство	створити файл	підкормити
19.	Рибне господарство	видалити файл	виловити
20.	Рибне господарство	переглянути файл	відібрати
21.	Лісове господарство	створити каталог	Посадити
22.	Лісове господарство	створити файл	обрізати
23.	Лісове господарство	видалити файл	зрубити
24.	Лісове господарство	переглянути файл	простукати
25.	Транспорт	створити каталог	Спроектувати
26.	Транспорт	створити файл	зібрати
27.	Транспорт	видалити файл	замінити
28.	Транспорт	переглянути файл	Оглянути
29.	Медицина	створити файл	вакцінувати

2.4 Робота з файлами через перенаправлення вхідних/вихідних потоків

Документуючі рішення цього розділу, у файлі *README.md* необхідно вказати наступний текстовий рядок у вигляді заголовку 3-го рівня *Markdown*-форматування:

«3 Робота з файлами через перенаправлення вхідних/вихідних потоків». В подальшому за результатами рішень кожного пункту завдання до файлу *README.md* додати знімки екрану та коментарі з урахуванням рекомендацій пункту 2.1.2.

Знаходячись в каталозі «*Laboratory-work-3*» *Git*-репозиторію, виконайте наступну послідовність завдань зі створення файлів через перенаправлення вхідних/вихідних потоків.

2.4.1 Створити файл з назвою як транслітерація вашого прізвища з прикінцевою цифрою 1, наприклад *blazhko_1*, використовуючи команду *cat* з перенаправленням *stdin*-потoku на *stdout*-потік так, що файл містив один рядок з вашими прізвищем та ім'ям.

2.4.2 Додати до створеного файлу через перенаправлення *stdout*-потoku ще один рядок з назвою вашої групи.

2.4.3 Створити файл з назвою як транслітерація вашого імені з прикінцевою цифрою 2, наприклад *blazhko_2*, який містить два рядки, створені через перенаправлення *stdout*-потoku двох наступних команд:

- команда визначення назви поточного каталогу, в якому ви знаходитесь, формує перший рядок;
- команда визначення імені поточного користувача ОС, формує другий рядок;

2.4.4 Об'єднати два раніше створені файли в один файл командою *cat* зі створенням нового файлу, назва якого – транслітерація вашого прізвища та імені із суфіксом-розширенням *.cat.txt*;

2.4.5 Повторити об'єднання файлів, але вже командою *paste* зі створенням нового файлу, де назва файлу – транслітерація вашого прізвища та імені із суфіксом-розширенням *.paste.txt*

2.4.6 В попередньому розділі та в цьому розділі було виконано завдання, які створювали файли у каталозі *Git*-репозиторію. Ці файли поки що мають статус неконтрольованих файлів, тому необхідно виконати *Git*-команди *add* та *commit* із коментарем «*Changed by Local Git*» для створення нового *Git*-знімку (нової *Git*-версії).

Команду *push* для перенесення змін з *Git*-репозиторію до *GitHub*-репозиторію треба буде зробити пізніше, вже під час виконання рекомендацій, наданих в розділі 2.6.

2.5 Проста обробка результатів виконання команд

Документуючі рішення цього розділу, у файлі *README.md* необхідно вказати наступний текстовий рядок у вигляді заголовку 3-го рівня *Markdown*-форматування:

«4 Проста обробка результатів виконання команд». В подальшому за результатами рішень кожного пункту завдання до файлу *README.md* додати знімки екрану та коментарі з урахуванням рекомендацій пункту 2.1.2.

Знаходячись в каталозі «*Laboratory-work-3*» *Git*-репозиторію, виконайте наступну послідовність завдань, поступово створюючи конвеєр команд.

2.5.1 Отримати перелік каталогів для швидкого пошуку файлів, які можуть виконуватися в командному рядку без вказування повного шляху до файлу, враховуючи, що кожен назву каталогу необхідно вивести в окремому рядку, наприклад, рядок з каталогами */c/user1:/c/user2/* необхідно перетворити на два окремі рядки */c/user1/* та */c/user1/*;

2.5.2 Змінити рішення попереднього завдання так, щоб всі назви каталогів були в окремих рядках, наприклад, рядок з каталогами */c/user1:/c/user2/* необхідно перетворити на чотири окремі рядки *c, user1, c, user2*

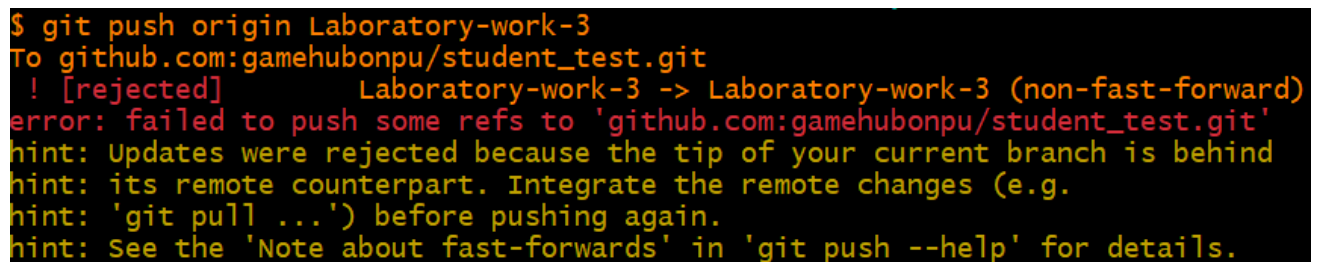
2.5.3 Змінити рішення попереднього завдання, впорядкувавши значення назв каталогів за зростанням та видаливши всі дублікати цих назв;

2.5.4 Змінити рішення попереднього завдання, визначивши лише перші 5 назв каталогів;

2.5.5 Провести статистичний аналіз результату завдання 2.5.3, отримавши кількість каталогів та розмір найбільшої назви каталогу.

2.6 Двонаправлене узгодження *Git*-репозиторія та *GitHub*-репозиторія

В розділі 2.4 було виконано завдання зі створення нового *Git*-знімку (нової *Git*-версії). Залишилося лише виконати команду *push* для перенесення змін з *Git*-репозиторію до *GitHub*-репозиторію. Але перед виконанням команди *push* треба враховувати, що в цей момент *GitHub*-репозиторій також вже оновив файл *README.md*. Тому таке завчасне виконання команди *push* може призвести до помилки, приклад якої представлено на рисунку 10. На рисунку видно, що *Git*-система відмовилася (*rejected*) оновлювати *GitHub*-репозиторій.



```
$ git push origin Laboratory-work-3
To github.com:gamehubonpu/student_test.git
! [rejected]        Laboratory-work-3 -> Laboratory-work-3 (non-fast-forward)
error: failed to push some refs to 'github.com:gamehubonpu/student_test.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Рис. 10 – Фрагмент екрану з прикладом помилки команди *push*

Аналіз коментаря до помилки може показати, що *Git*-система виявила *Git*-конфлікт одночасного редагування файлів у двох репозиторіях та пропонує вирішити виявлену проблему через повторне оновлення локального *Git*-репозиторію змінами з *GitHub*-репозиторію, використовуючи команду *pull*:

```
git pull origin Laboratory-work-3
```

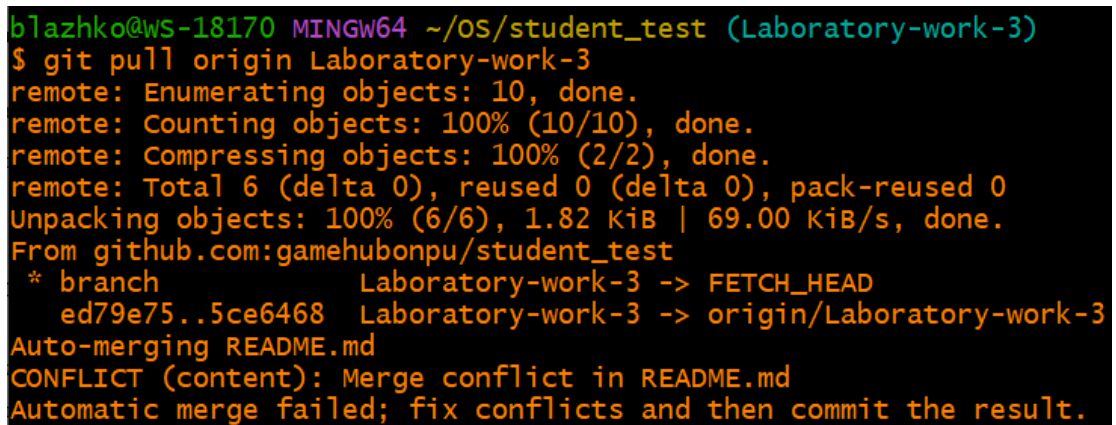
В реальних ІТ-проектах не рекомендується одночасно змінювати файли одної гілки в різних репозиторіях, бо кожний учасник проекту має можливість створити свою гілку для власних змін. Але якщо така зміна відбувається, тоді під час виконання оновлення через команду *pull* *Git*-система про це дізнається та перерве цей процес, щоб учасник, який це зробив, надав свої пояснення стосовно мети змін.

Для виявлення подібного *Git*-конфлікту проведемо наступний експеримент з одночасною зміною файлу *README.md*:

– у *GitHub*-репозиторії до файлу додано рядок *Changed by GitHub*;

– у локальному *Git*-репозиторії до файлу додано рядок *Changed by Local Git*.

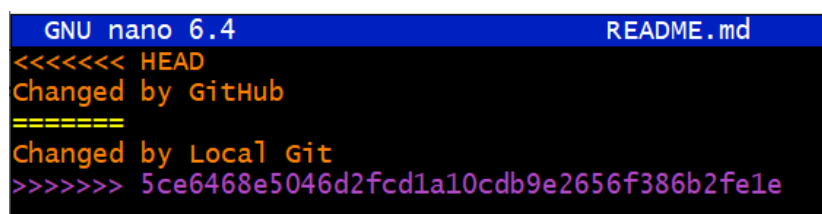
Після виконання команди *git pull* виявила *Git*-конфлікт одночасного оновлення файлу *README.md*, як показано на рисунку 11.



```
blazhko@ws-18170 MINGW64 ~/OS/student_test (Laboratory-work-3)
$ git pull origin Laboratory-work-3
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), 1.82 KiB | 69.00 KiB/s, done.
From github.com:gamehubonpu/student_test
* branch      Laboratory-work-3 -> FETCH_HEAD
  ed79e75..5ce6468 Laboratory-work-3 -> origin/Laboratory-work-3
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Рис. 11 – Фрагмент екрану з прикладом виявлення конфлікту

Для розв'язання конфлікту одночасного оновлення файлу *README.md* система автоматично оновлює зміст цього файлу на локальному *Git*-репозиторії, приклад якого показано на рисунку 12.



```
GNU nano 6.4 README.md
<<<<<<< HEAD
Changed by GitHub
=====
Changed by Local Git
>>>>>> 5ce6468e5046d2fcd1a10cdb9e2656f386b2fe1e
```

Рис. 12 – Фрагмент екрану з прикладом вмісту файлу *README.md* із конфліктним вмістом

Рядок із символами ===== розділяє вміст двох версій файлу. Для розв'язання конфлікту необхідно залишити лише ті рядки, які будуть остаточними у файлі. Також треба видалити символи <<<<, >>>> =====.

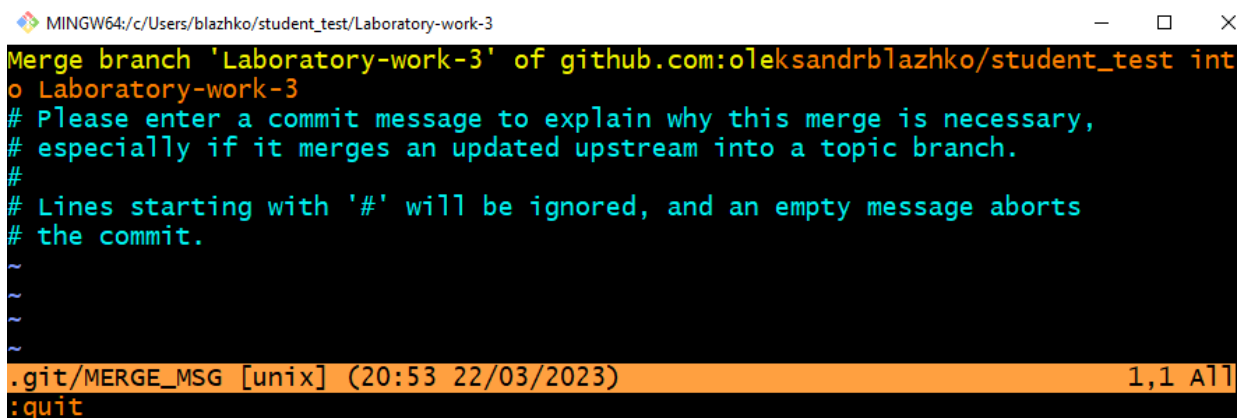
Після завершення редагування файлу *README.md* необхідно виконати команди *add*, *commit* та *pull*.

Після виконання оновлення *Git*-репозиторію вже можна виконати команду *push*:

```
git push origin Laboratory-work-3
```

В процесі виконання команди *push* на *GitHub*-сервері розпочнеться підготовка до процесу злиття нової гілки та основної гілки (*main* або *master*) репозиторію. *GitHub*-сервер може запросити вказати повідомлення через редагування спеціального файлу в редакторі *vi*, де користувач може вказати причини виконання таких змін, як це показано на рисунку 13. Але користувачу достатньо залишити цей файл без редагування та вийти з цього процесу через послідовність наступних клавіш редактору *vi* (рисунок 13):

- символ : (двокрапка) – для переходу до режиму введення команд редактору;
- команда *quit*



```
MINGW64: c:/Users/blazhko/student_test/Laboratory-work-3
Merge branch 'Laboratory-work-3' of github.com:oleksandrblazhko/student_test into Laboratory-work-3
# 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.
~
~
~
~
.git/MERGE_MSG [unix] (20:53 22/03/2023) 1,1 All
:quit
```

Рис. 13 – Фрагмент екрану з прикладом редагування спеціального файлу в редакторі *vi*

2.7 Особливості надання рішень завдань лабораторної роботи на перевірку

Всі ваші рішення завдань лабораторної роботи було розміщено в окремій (тимчасовій) гілці *GitHub*-репозиторію. Традиційно зміст окремої гілки треба перенести до основної гілки, виконавши операцію злиття (*merge*) цих гілок.

Перед злиттям гілок традиційно проводиться перевірка змісту змін у гілці через процес *Code Review* на наявність помилок. Така перевірка виконується не авторами змін, а іншими учасниками проекту – рецензентами (*Reviewer*). В умовах навчального процесу таким учасником є викладач, який виконує процес *Code Review* для подальшого оцінювання рішень та можливого повернення цих рішень на доопрацювання студентами.

Для початку процесу *Code Review* автор змін виконує запит *Pull Request*, перейшовши у розділ «*Pull Request*» верхнього меню *GitHub*-репозиторію. На рисунку 14 наведено фрагмент екрану з ініціалізацією запиту *Pull Request*, на якому на жовтому фоні вказано про операцію *push*, яку раніше виконано з гілкою.

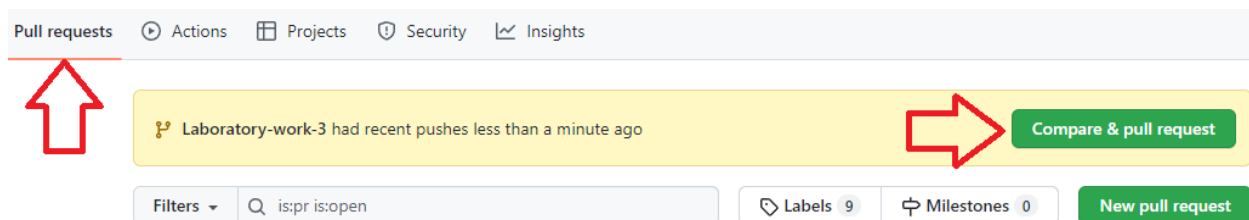


Рис. 14 – Фрагмент екрану з ініціалізацією запиту *Pull Request*

На рисунку 15 наведено фрагмент екрану з налаштування запиту *Pull Request*, який визначає наступні дії:

- вказати назву запиту *Pull Request* – «*Laboratory work 3*»;
- вибрати зі списку рецензентів учасника з обліковим записом вашого викладача, наприклад, *oleksandrblazhko* або *Miroslavdr*
- вибрати зі списку ініціаторів учасника з вашим обліковим записом.

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

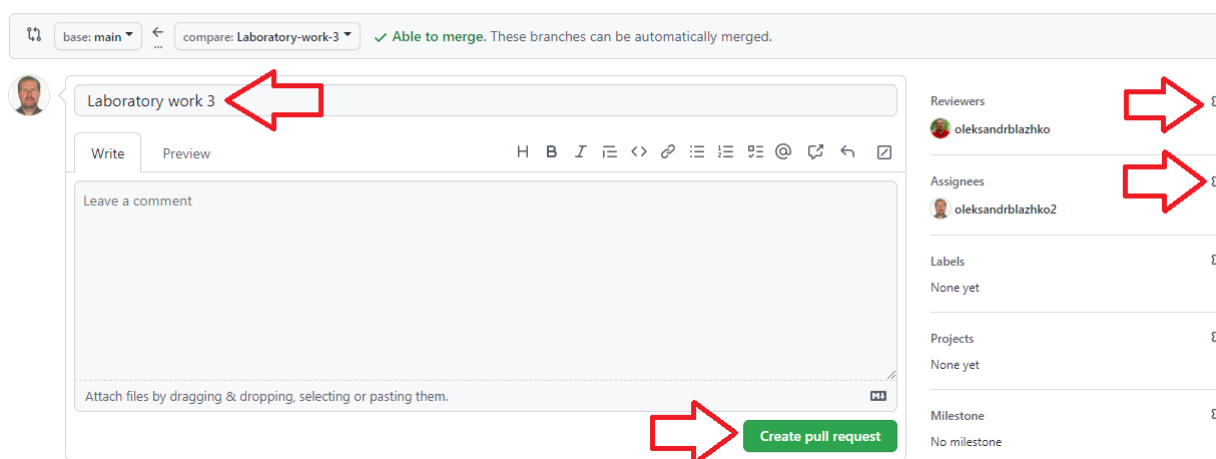






Рис. 15 – Фрагмент екрану з налаштування запиту *Pull Request*

Після виконання запиту з'явиться екранна форма з процесом початку злиття змін, як показано на рисунку 16.


Laboratory work 3 #5



 **Open** oleksandrblazhko2 wants to merge 2 commits into `main` from `Laboratory-work-3` 



Conversation 0 Commits 2 Checks 0 Files changed 1



 oleksandrblazhko2 commented now 

No description provided.


 oleksandrblazhko2 and others added 2 commits 15 minutes ago

-  Changed by Local Git 32e0869
-  Update README.md Verified f115f18

  oleksandrblazhko2 requested a review from oleksandrblazhko now

  oleksandrblazhko2 self-assigned this now

Add more commits by pushing to the `Laboratory-work-3` branch on `oleksandrblazhko/student_test`.






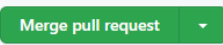

-  **Review requested** Show all reviewers
Review has been requested on this pull request. It is not required to merge. [Learn more.](#)
-  1 pending reviewer
-  **This branch has no conflicts with the base branch**
Merging can be performed automatically.
-  **Merge pull request**  You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

Рис. 16 – Фрагмент екрану з налаштування запиту *Pull Request*

Для реальних ІТ-проектів у *GitHub*-репозиторіях найчастіше автор змін не може натиснути кнопку «*Merge pull request*» (цю кнопку буде заблоковано та представлено червоним кольором) поки рецензенти не нададуть таке право. Але у навчальному репозиторії, який, перш за все, має тип *Private*, така можливість є, тому:

Увага! Не натискайте кнопку «*Merge pull request*»!

Це повинен зробити лише рецензент, який є вашим викладачем!

Коли рецензент-викладач перегляне ваше рішення він виконає злиття нової гілки та основної гілки, натиснувши кнопку «*Merge pull request*». Якщо рецензент знайде помилки, він повідомить про це у коментарях, які з'являться на сторінці *Pull request*.

Подробиці про особливості процесів *Code Review* у промислових умовах наведено в документі [2] списку літератури.

3 Оцінка результатів виконання завдань лабораторної роботи

Оцінка	Умови
+3 бали	1) всі рішення відповідають завданням 2) <i>Pull Request</i> представлено не пізніше найближчої консультації після офіційного заняття із захисту лабораторної роботи
-0.5 балів за кожну помилку	в рішенні є помилка, про яку вказано в <i>Code Review</i>
-1 бал	<i>Pull Request</i> представлено пізніше дати найближчої консультації після офіційного заняття із захисту лабораторної роботи за кожний тиждень запізнення
+2 бали	Отримано правильну відповідь на два запитання, які стосуються призначення команд, представлених на знімках екранів рішень завдань роботи

Контрольні запитання

1. Як розшифровується абревіатура *DOS*?
2. Що таке файл?
3. Що таке каталог?
4. Опишіть особливості файлових систем з ієрархічною структурою.
5. В чому різниця між *Bash*-командою та утилітою? Як можна дізнатися про таку різницю?
6. Для чого в ОС використовується змінна оточуючого середовища *PATH*?
7. Переглядаючи кожний каталог командою *ls*, можна побачити два прихованих файли з назвою крапка зі слешем та дві крапки зі слешем. Що це за файли?
8. В чому різниця між командою *pushd* та командою *cd* ?
9. Що нового можна дізнатися про файли через команду *ls* з опцією *l* ?
10. Що користувачеві ОС надає команда *alias* ?
11. Що таке перенаправлення потоків даних?
12. Чим *stdout*-потік відрізняється від *stdin*-потіку?
13. Наведіть приклади пристроїв комп'ютера, які використовують *stdout*-потік?
14. Наведіть приклади пристроїв комп'ютера, які використовують *stdin*-потік?
15. В чому полягає принцип абстрагування від апаратних компонент?

16. Що під час перенаправлення потоків означає комбінація символів `2>` (двійка та більше) ?

17. В чому різниця між одним символом більше або менше та двома символами більше або менше?

18. Як розшифровується скорочена назва для команди *cat* ? Яку ще задачу, крім вказаної у назві, команда може виконувати?

19. В чому різниця між командою *paste* та командою *cat*?

20. Як розшифровується скорочена назва для команди *tr* ? Які завдання вона може виконувати?

21. Про що говорить комбінація символів `\n` (слеш та *n*) під час роботи із рядками тексту?

22. Що таке конвеєр команд?

23. Що таке неіменованний канал?

24. Яка перевага використання конвеєру завдяки буферизації?

25. Як розшифровується скорочена назва для команди *wc* ? Які завдання вона може виконувати?

26. *Bash*-оболонка має різні команди, які можуть виконувати схожі дії. Які є рекомендації стосовно кращого вибору команд?

27. Для чого в *Git*-системах використовується запит *Pull Request*?

28. Чому виникає *Git*-конфлікт ?

29. Як можна розв'язати *Git*-конфлікт ?

Література

1 Олександр Блажко. Проста обробка текстових даних в оболонці команд *Unix*-подібних ОС. Відео-запис лекції. URL : <https://www.youtube.com/watch?v=UdslnE6xgEw>

2 Олександр Блажко. Особливості проведення *CodeReview* в *GitHub*-репозиторіях. URL : <https://drive.google.com/file/d/15iYHMr3Lzr04Tk0EWo9qKUOs5UArohV1>