Изображение выглядит как текст

Автоматически созданное описание

**Отсчет по лабораторным работам**

по дисциплине «Системное программное обеспечение GNU/Linux»

Студент гр. 5130904/40008 Лиходиевский А. М.

Руководитель доц. Шмаков В. Э.

Оглавление

[Лабораторная работа №1 3](#__RefHeading___Toc153_2440990709)

[Цель работы 3](#__RefHeading___Toc7506_2440990709)

[Последовательность выполнения работы: 3](#__RefHeading___Toc7508_2440990709)

[Вывод 9](#__RefHeading___Toc318_3492897412)

[Лабораторная Работа №2 10](#__RefHeading___Toc320_3492897412)

[Цель работы 10](#__RefHeading___Toc322_3492897412)

[Последовательность выполнения работы 10](#__RefHeading___Toc324_3492897412)

[ЛАБОРАТОРНАЯ РАБОТА №3 16](#__RefHeading___Toc906_2500951832)

[Цель работы: 16](#__RefHeading___Toc2221_714890939)

[Вывод: 27](#__RefHeading___Toc2225_714890939)

[ЛАБОРАТОРНАЯ РАБОТА №4 28](#__RefHeading___Toc913_1721718268)

[Цель работы. 28](#__RefHeading___Toc6341_1721718268)

[Выполнение работы 28](#__RefHeading___Toc1376_2152403033)

[Вывод. 30](#__RefHeading___Toc6339_1721718268)

[Лабораторная работа №5 31](#__RefHeading___Toc3778_1517724926)

[Цель работы 31](#__RefHeading___Toc3780_1517724926)

[Выполнение работы: 31](#__RefHeading___Toc10763_3794621434)

[ЛАБОРАТОРНАЯ РАБОТА № 6 37](#__RefHeading___Toc10890_3794621434)

[Цель работы. 37](#__RefHeading___Toc10892_3794621434)

[Выполнение работы. 37](#__RefHeading___Toc11160_3794621434)

[Лабораторная №7 42](#__RefHeading___Toc3782_1517724926)

[Цель работы 42](#__RefHeading___Toc3784_1517724926)

[Выполнение работы 42](#__RefHeading___Toc3786_1517724926)

[ЛАБОРАТОРНАЯ РАБОТА № 8 48](#__RefHeading___Toc11250_2603556727)

[Цель работы. 48](#__RefHeading___Toc11252_2603556727)

[Последовательность выполнения работы: 48](#__RefHeading___Toc11254_2603556727)

[Лабораторная работа №9 54](#__RefHeading___Toc1378_2152403033)

[Цель работы 54](#__RefHeading___Toc1380_2152403033)

[Последовательность выполнения работы 55](#__RefHeading___Toc1382_2152403033)

[Лабораторная работа №10 67](#__RefHeading___Toc12302_2603556727)

[Цель работы 67](#__RefHeading___Toc12304_2603556727)

[Последовательность выполнения работы 67](#__RefHeading___Toc12306_2603556727)

[ЛАБОРАТОРНАЯ РАБОТА №11 71](#__RefHeading___Toc12308_2603556727)

[Цель работы 71](#__RefHeading___Toc12310_2603556727)

[Последовательность выполнения работы 71](#__RefHeading___Toc12312_2603556727)

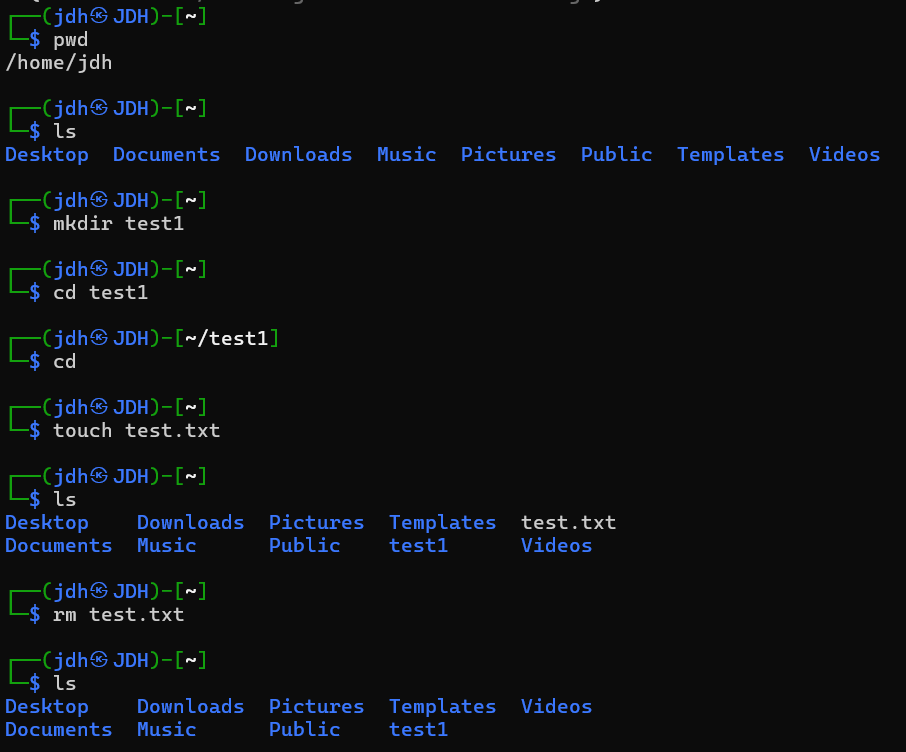
# **Лабораторная работа №1**

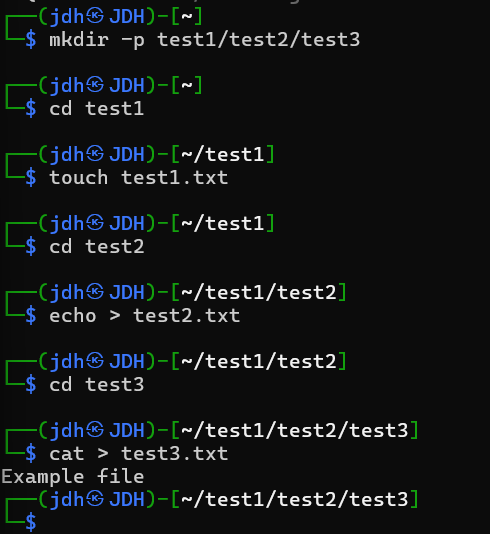
**Базовые команды ОС Linux**

## **Цель работы**

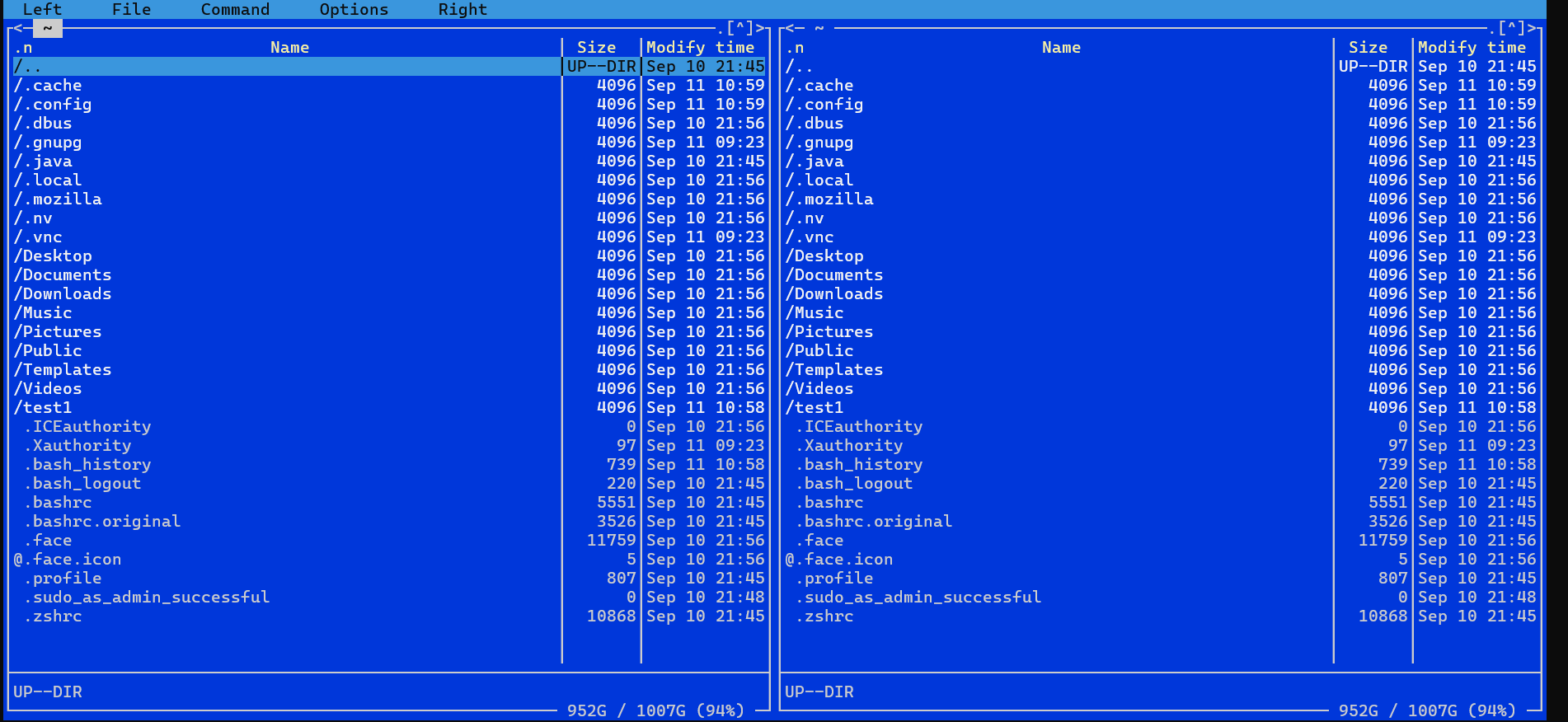
Освоение минимального набора базовых команд операционной системы Linux, знакомство с файловой системой, особенностями прав доступа, получение первичных навыков работы под Linux.

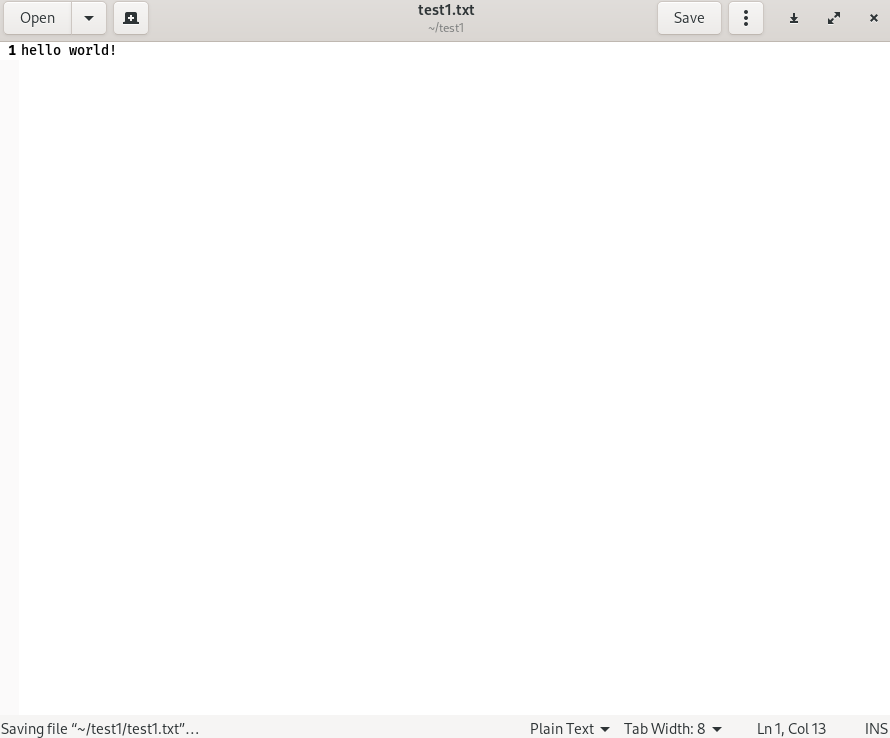
## **Последовательность выполнения работы:**

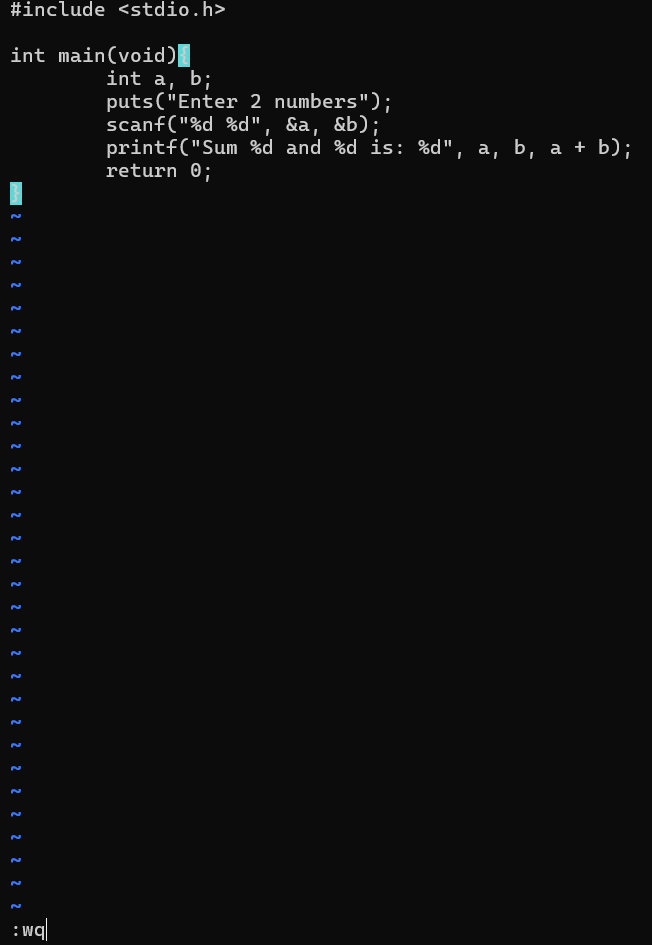
1. Запустил терминал и выполнил серию команд shell, такие, как pwd , ls , cd , mkdir , rm
2. 



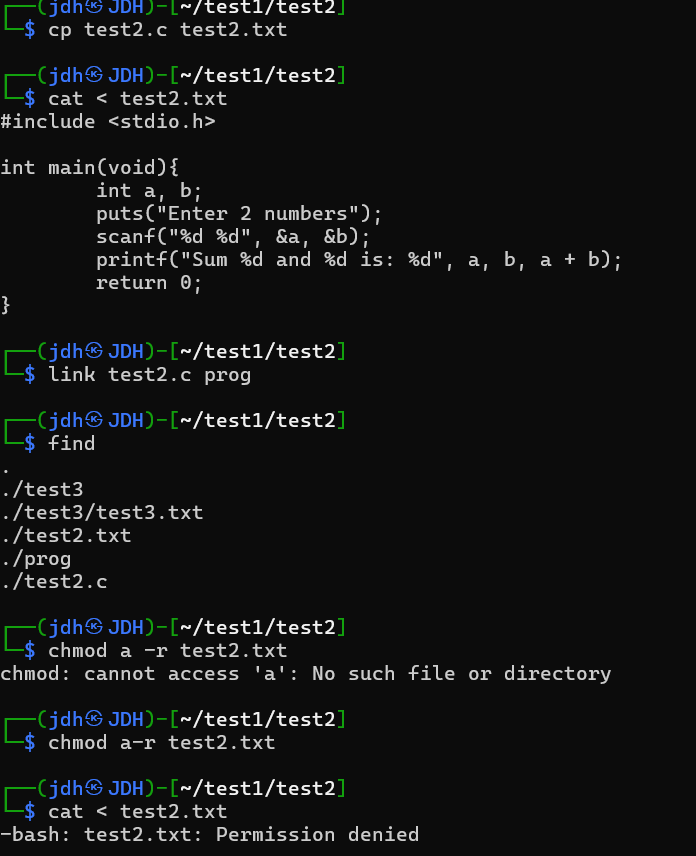
4.

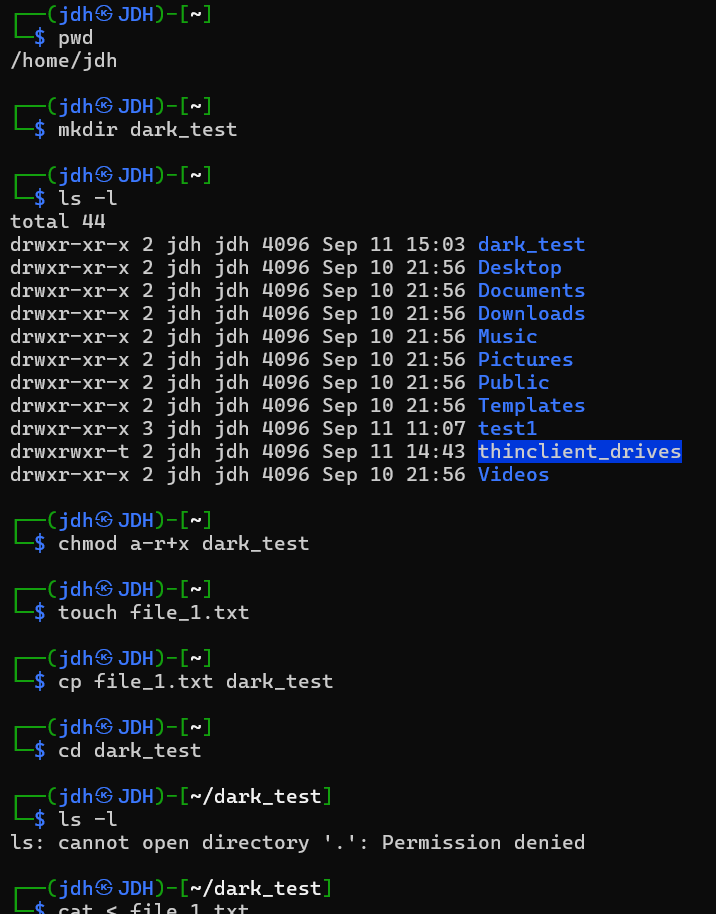






5.



6.

## **Вывод**

Было произведено освоение минимального набора базовых команд операционной системы Linux, знакомство с файловой системой, особенностями прав доступа, получение первичных навыков работы под Linux.

# Лабораторная Работа №2

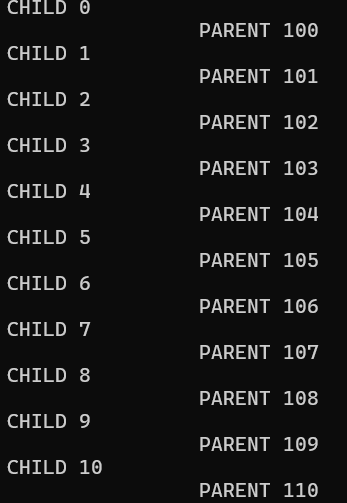
## Цель работы

Знакомство с характерной для Linux схемой порождения и завершения процессов, с отношениями типа потомок – родитель, со способами передачи информации о событии завершения процесса.

## **Последовательность выполнения работы**

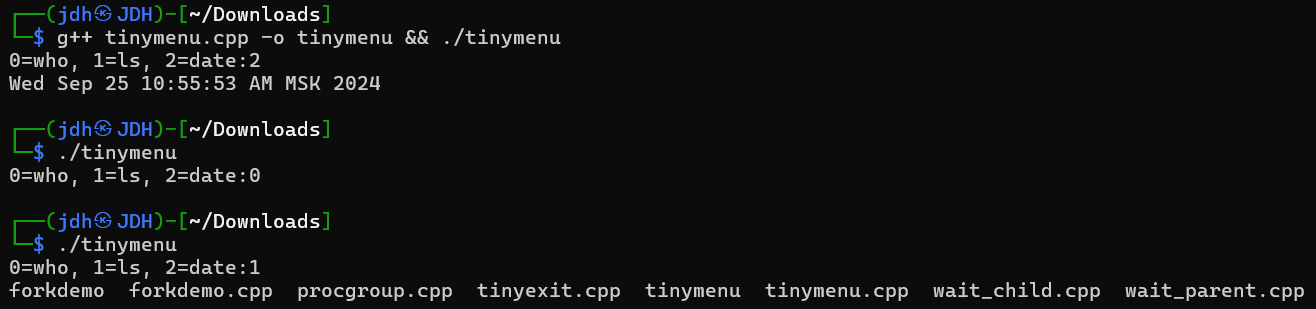
1. Запуск и компиляция файлов:

1) forkdemo.cpp



Функция fork() создаёт точную копию текущего процесса (дочерний процесс), которая выполняется параллельно родительскому, поэтому в выводе вперемешку присутствует вывод CHILD и PARENT. Причём, если запустить программу ещё раз, вывод будет отличаться (по-разному будет перемешиваться вывод CHILD и PARENT).

2) tinymenu.cpp

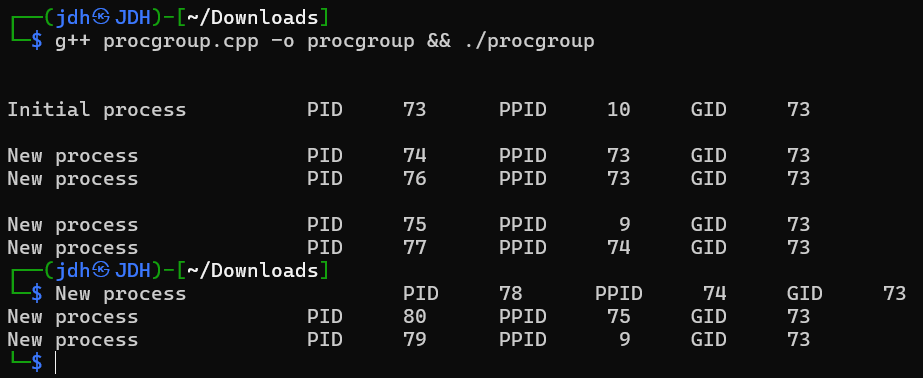


Функция execlp вызывает выбранную функцию из массива cmd.

3) tinyexit.cpp

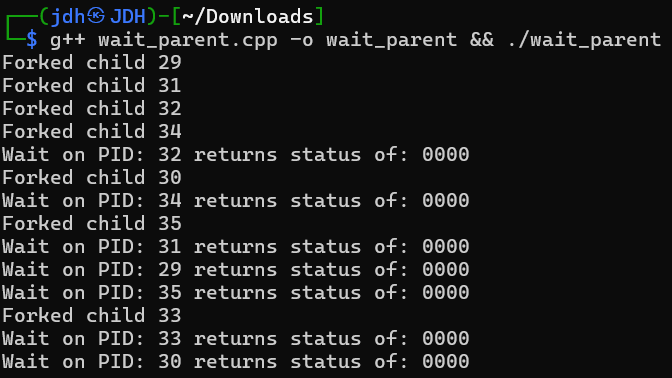
Программа (так же, как и предыдущая) выполняет указанную команду, но не завершается после каждой выполненной команды.

4) procgroup.cpp



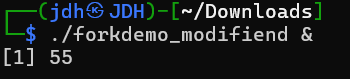
Программа генерирует копии процессов.

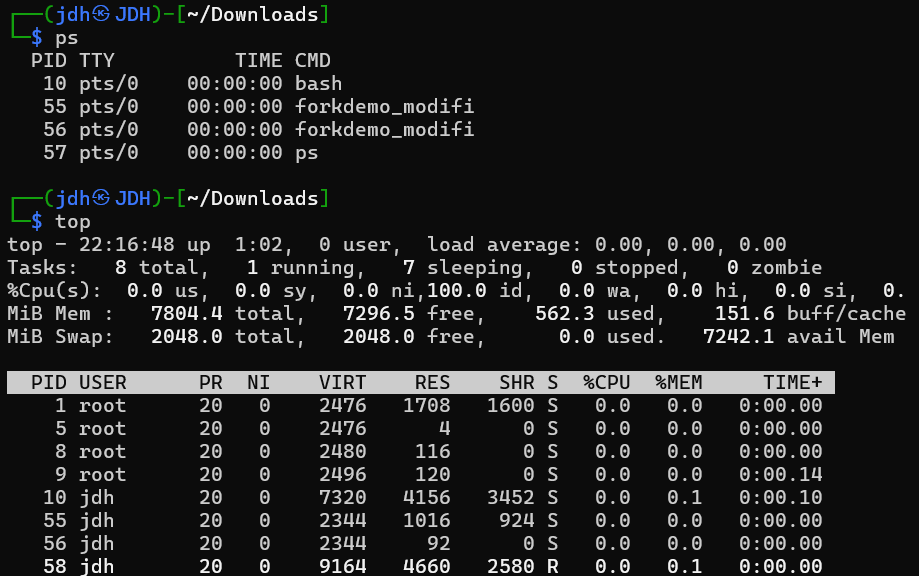
5) wait\_parent.cpp

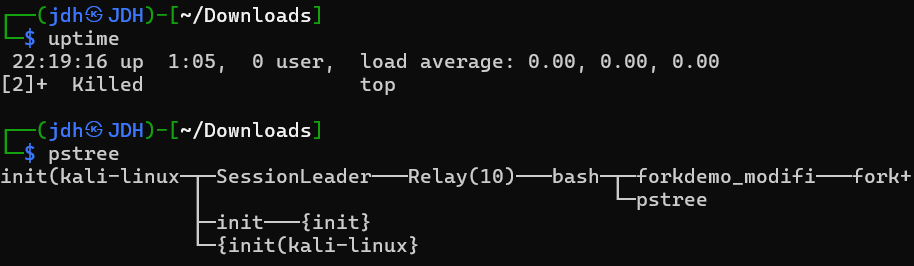


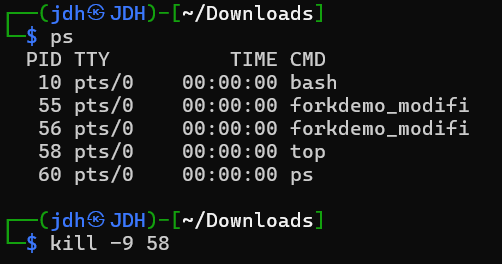
Процесс wait\_parent запускает процесс wait\_child, а затем ожидает завершения своих дочерних процессов.

3. Модифицируйте программу forkdemo.cpp (или создайте собственную), так чтобы ввод/вывод на терминал отсутствовал, а при проходе по циклу была временная задержка, например sleep (7). Запустите эту программу в фоновом режиме (background), введя при запуске символ & после пробела и зафиксировав значение PID, назначенное системой фоновому процессу при запуске. Выполните на терминале команды ps, top, uptime, pstree. Снимите свой фоновый процесс командой kill с соответствующими параметрами.









Используемые команды:

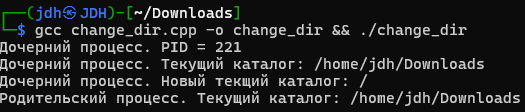
1. ps позволяет получить информацию о процессах и работать с ними
2. top выводит список работающих в системе процессов и информацию о них в отсортированном по нагрузке на процессор виде (причём в реальном времени).
3. uptime даёт информацию о: текущем времени, онлайн-пользователях, длительности работы системы и средней загрузке системы.
4. pstree показывает запущенные процессы в виде дерева.

4. Исследуйте, что произойдет, если процесс-потомок сменит

текущий каталог, будет ли изменен текущий каталог для родителя? Создайте программу, подтверждающую ответ и приведите в отчёте.

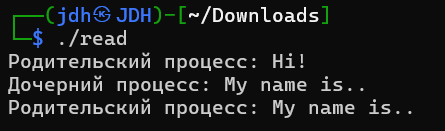
Если процесс-потомок сменит текущий каталог, текущий каталог для родителя изменяться не будет. Подтвердим предположение с помощью специально разработанного кода.

Результат запуска:



Процесс — родитель не изменил текущий каталога

5. Проиллюстрируйте как процесс-родитель и процесс-потомок разделяют один и тот же дескриптор и смещение текстового файла. Для этого составьте программу, в которой процесс-родитель должен открывать текстовый файл и запускать потомка. Потомок должен читать порцию данных из открытого файла и выводить на консоль. По завершению потомка родитель должен читать из того же файла и выводить результат на консоль. Можете использовать вызов sleep() для синхронизации доступа родителя и потомка к файлу.



Вывод: в ходе данной лабораторной работе был получен опыт работы с процессами и программировании на Си в среде Linux

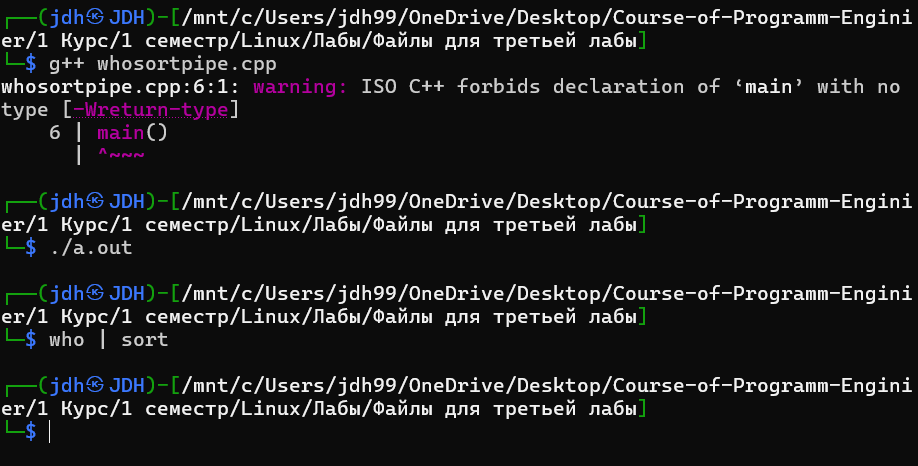
# ЛАБОРАТОРНАЯ РАБОТА №3

## Цель работы:

Знакомство с программными каналами

Программные каналы

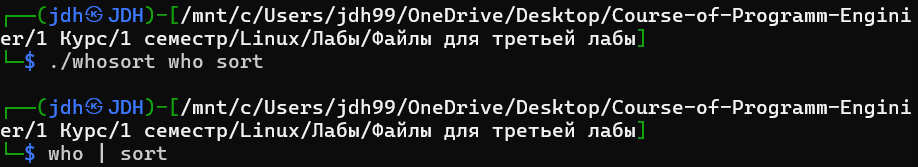
1. Скомпилируйте и выполните программу whosortpipe.cpp. Сопоставьте результат выполнения программы с выполнением этих же двух команд с терминала в конвейерном режиме (с использованием <>). Анализ результатов работы этой программы (как и всех последующих) с соответствующими скриншотами приведите в отчете.



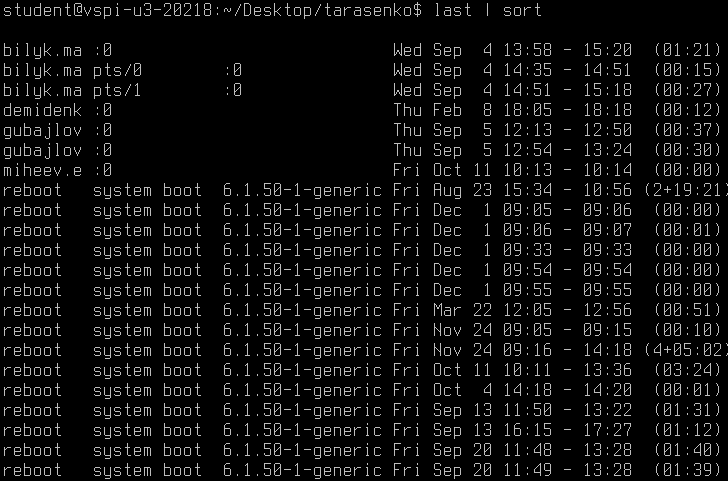
Результаты абсолютно одинаковы.

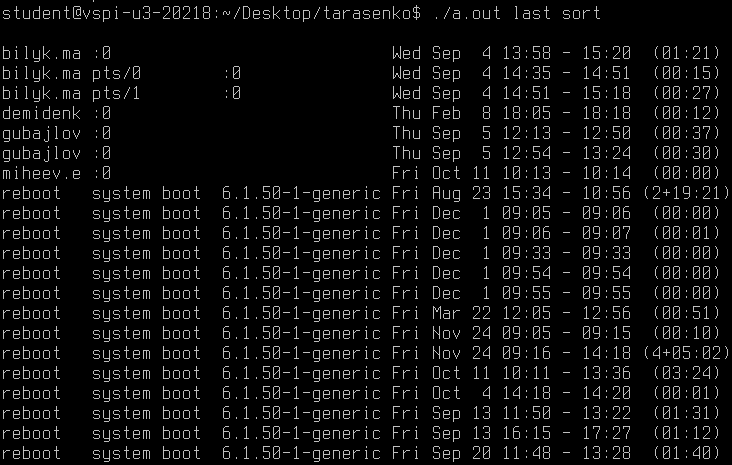
2. Программу cmdpipe.cpp запускайте после компиляции, задавая ей при стартах в качестве параметров командной строки пары команд shell для конвейеризации (who и sort ; last и sort ; last и more ; pstree и more). Сопоставьте результаты запусков программы с выполнением тех же пар командиз shell в конвейерном режиме.

2.2 Сравнение запуска с who sort:

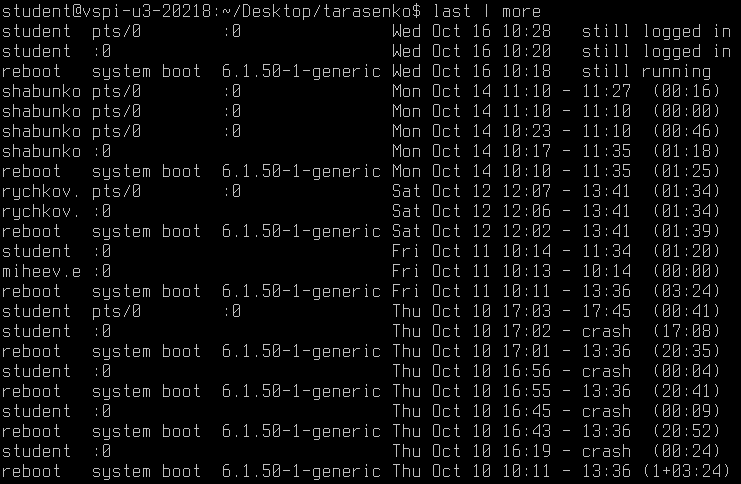


2.3 Сравнение запуска с last sort:





2.4 Сравнение запуска с last more:



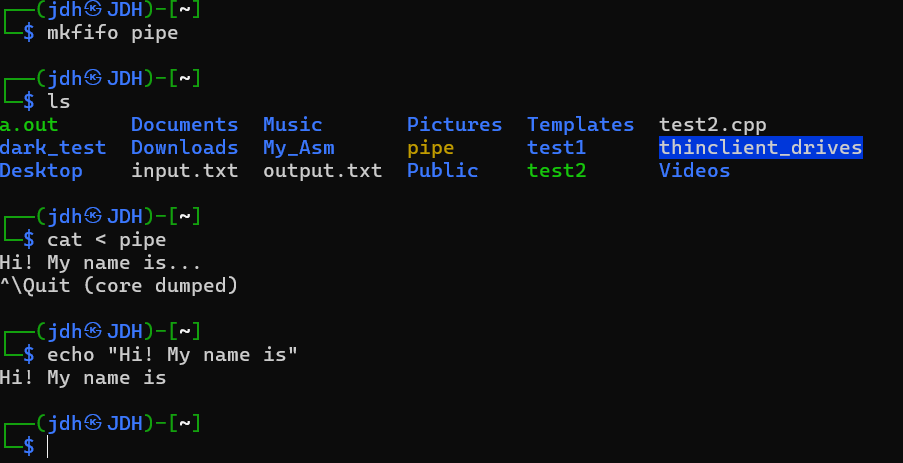
2.5 Сравнение запуска с pstree more:



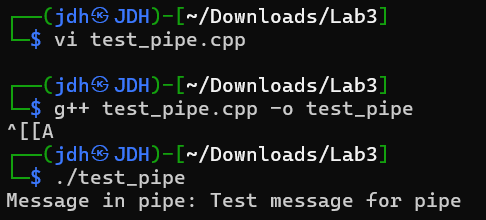


3. Создайте программный канал типа named pipe из командной строки (пример в лекциях). Используя необходимые системные вызовы организуйте канал named pipe в программе, сравните результат выполнения обмена по нему в программе с тем, чего можно достичь, создавая named pipe на терминале.

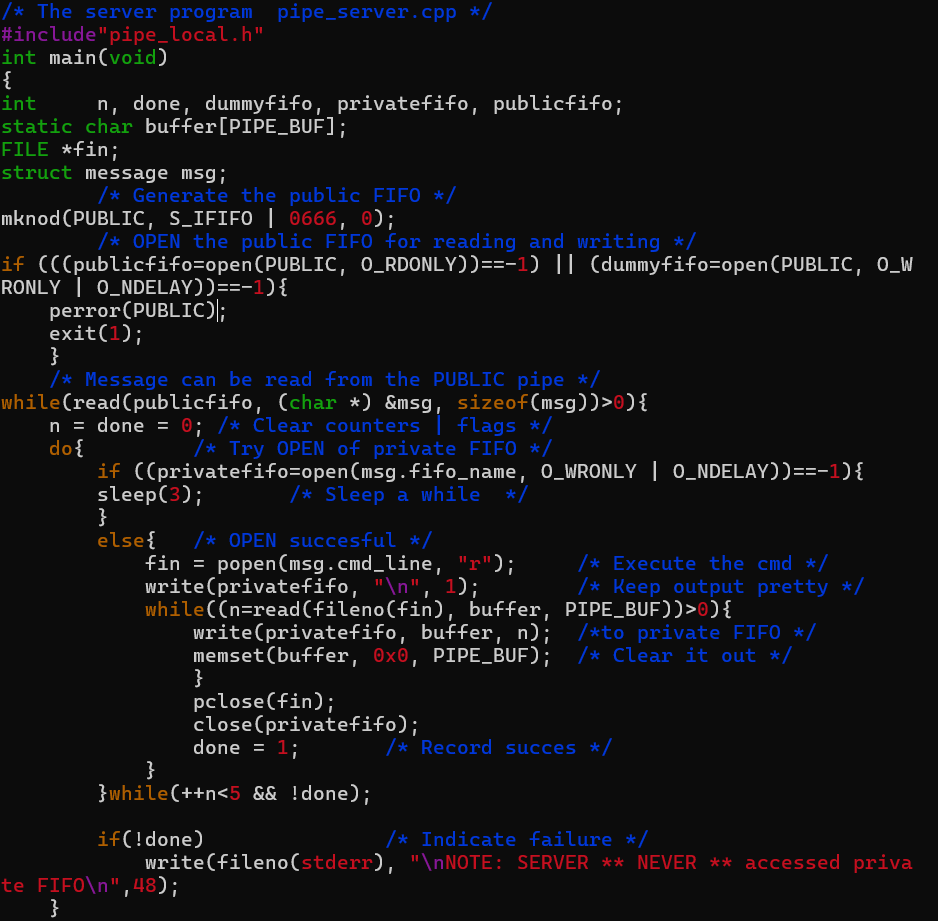
3.1 shell



3.2



4. Разберите и выполните пример клиент-серверного взаимодействия, организованного на конвейерах различного типа. Исходный текст примера содержится в файлах pipe\_server.cpp , pipe\_client.cpp и pipe\_local.h и разобран в материалах лекций. Сервер запускается в фоновом режиме. Проанализируйте какие конвейеры используются, как они создаются, как функционирует данная системы, ее недостатки. Программа-сервер этого примера исполняет каждый командный запрос поочередно. Если какой-либо запрос потребует много времени, все остальные клиентские процессы будут ожидать обслуживания. Коды программ были переписаны, так как прежние вызывали ошибки при вызове из терминала.





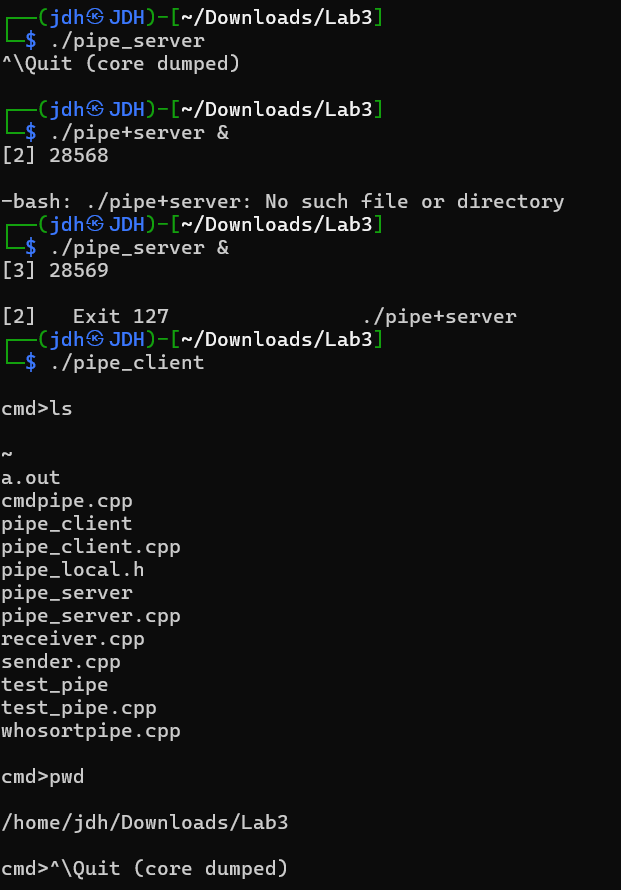
******

Схема взаимодействия

1. **Клиент**:
   * Создает приватный FIFO.
   * Открывает публичный FIFO для записи.
   * Вводит команды и отправляет их серверу через публичный FIFO.
   * Открывает приватный FIFO для чтения результатов.
   * Читает и отображает результаты на stderr.
   * Закрывает и удаляет приватный FIFO перед завершением работы.
2. **Сервер**:
   * Создает публичный FIFO.
   * Открывает публичный FIFO для чтения.
   * Ожидает сообщений от клиентов.
   * При получении сообщения пытается открыть приватный FIFO клиента.
   * Выполняет команду и отправляет результаты обратно клиенту через приватный FIFO.
   * Закрывает приватный FIFO после отправки результатов.

***Анализ конвееров:******Взаимодействие происходит через два типа FIFO:***

***Публичный FIFO:*** *Используется для приема командных запросов от клиентов.*

***Приватный FIFO:*** *Создается для каждого клиента, чтобы вернуть ему результаты выполнения команды.*

***Функционирование системы:***

***1. Создание FIFO:*** *Публичный канал, создается с помощью mkfifo. Приватные каналы создаются по запросу клиента и используются для передачи результатов выполнения команды.*

***2. Клиент*** *отправляет команду на сервер через публичный канал и ждет ответа в своем приватном канале.*

***3. Сервер*** *получает команду через публичный канал, выполняет её с помощью popen(), а затем отправляет результат обратно клиенту через его приватный канал.*

***Недостатки системы:***

***Последовательная обработка —*** *сервер выполняет запросы клиентов по очереди, из-за чего медленные запросы задерживают остальные.*

***Отсутствие многозадачности —*** *сервер не обрабатывает несколько запросов одновременно.*

***Постоянное создание каналов —*** *для каждого запроса клиента создается и удаляется приватный канал.*

5.Модифицируйте программу pipe\_server.cpp из предыдущего задания так, чтобы при получении нового сообщения от очередного клиента сервер порождал очередной дочерний процесс для выполнения задачи обслуживания данного запроса (исполнения переданной от клиента команды и переправки результата обратно клиенту).

## Вывод:

Мы использовали программные каналы для межпроцессорной коммуникации, увидели, какие недостатки и ограничения могут возникать при их применении. Также изучили простейшее клиент-серверное взаимодействие.

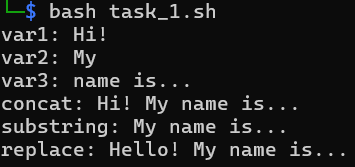
# ЛАБОРАТОРНАЯ РАБОТА №4

## Цель работы.

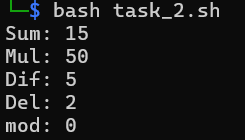
Знакомство с важным атрибутом любой операционной системы – переменными среды (или переменными окружения) environment variables и с возможностями их использования в Linux. Освоение языка для составления командных сценариев и написание набора полезных для системного администрирования скриптов.

## Выполнение работы

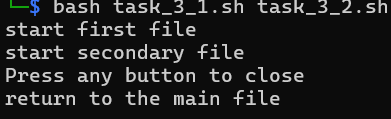
1. Создаем переменные в bash файле, выполняет конкатенацию строк, извлечение подстроки и заменяем значение в конкатенированной строке на произвольное



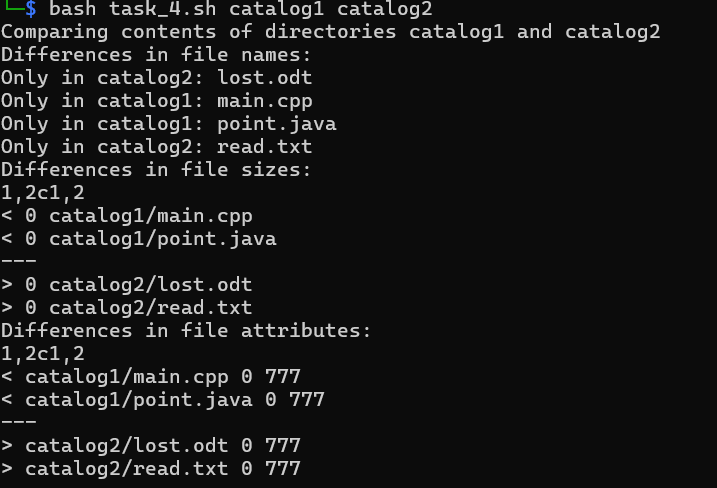
2. Создаем числовые переменные и выполняем с ними арифметические операции



3. Запускаем 2 процесса последовательно. Первый файл запускает второй, название которого получает как параметр при вызове, после выполнения второго файла возвращаемся в первый



4. Пишем скрипт для проверки различия в размерах и аттрибутах файлов в двух каталогах



5. Создаем командный файл, который принимает 2 аргумента: последовательность и имя результирующего файла, после чего запускаем. Программа ищет все текстовые файлы с заданной последовательностью и записывает его имя и номер строки с последовательностью в результирующий





#### Вывод.

В ходе выполнения лабораторной работы были созданы и протестированы Bash-скрипты для работы с переменными окружения, выполнения арифметических операций, взаимодействия двух файлов и сравнения содержимого каталогов.

# Лабораторная работа №5

## Цель работы

Манипуляции с правами доступа при создании в системе

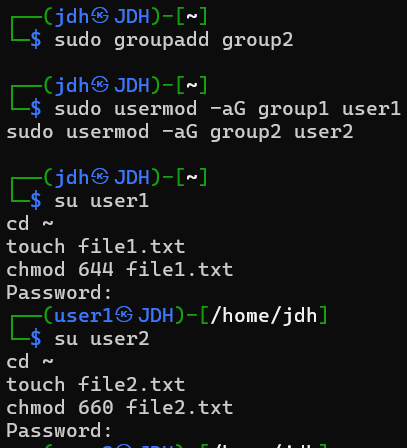
учетных записей и исследование влияния прав на файловые операции.

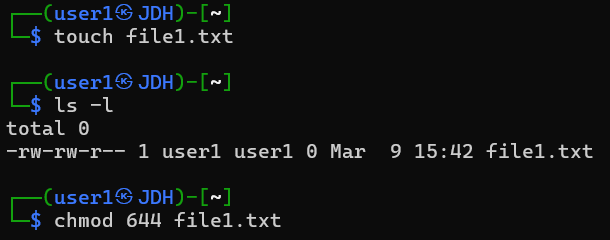
Изучение специфик фонового (background) и диалогового (foreground) режимов

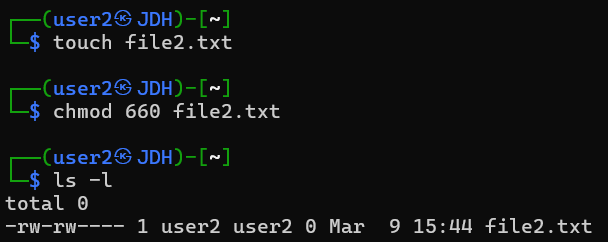
исполнения процессов и способов переключений между этими режимами.

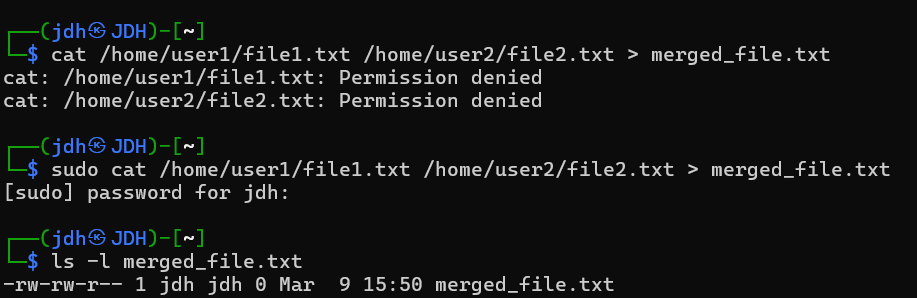
## Выполнение работы:

1.





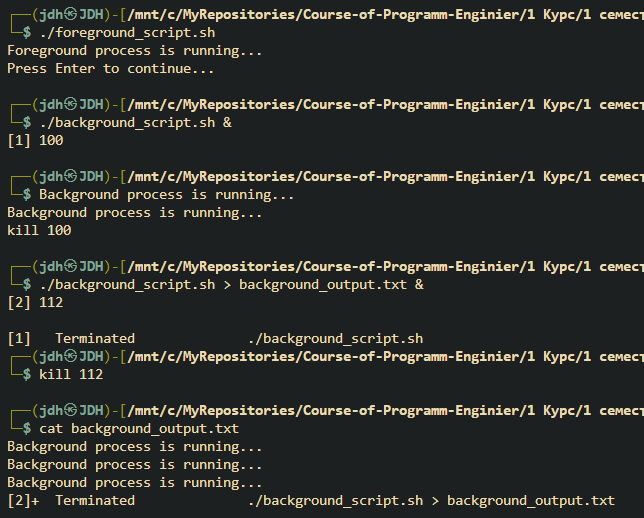




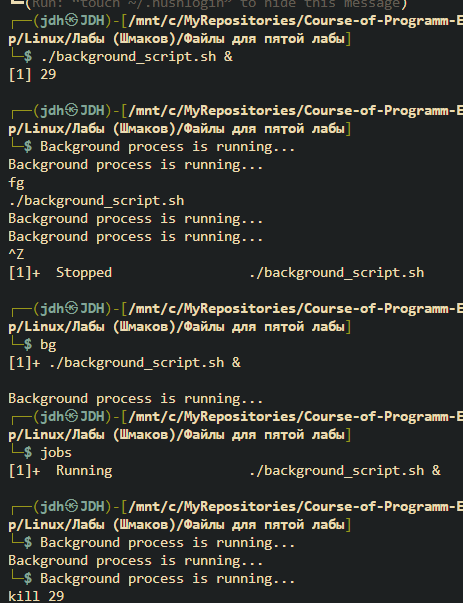
Права доступа на новый файл определяются **umask** (маской прав по умолчанию) текущей сессии или пользователя, который создаёт файл. Обычно umask по умолчанию для root — это 022, что означает, что новый файл будет создан с правами 644 (владелец — чтение и запись, группа и другие — только чтение).

Права доступа на file1.txt и file2.txt влияют только на то, сможешь ли ты их прочитать. Если у тебя нет прав на чтение этих файлов, ты не сможешь их объединить, даже с использованием sudo.

2.



3.



Анализ запуска:   
- запущенный в фоновом режиме процесс был переведём в диалоговый режим для ввода

- после ввода его вернули в фоновый

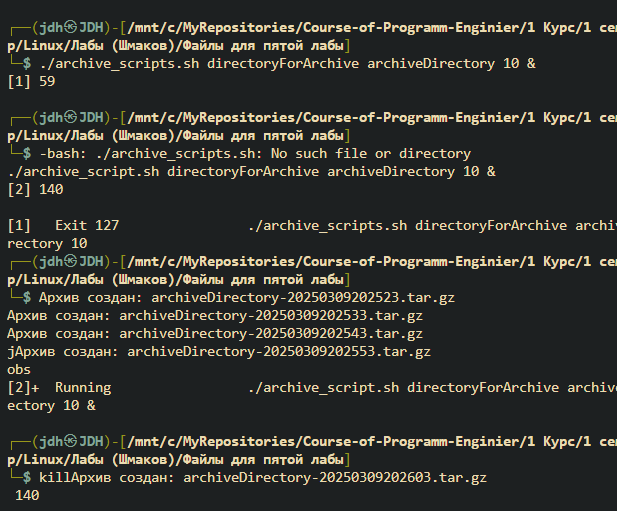
- в фоновом режиме процесс выводит результат в терминал (для наглядности)

- fg переводит в диалоговый режимам

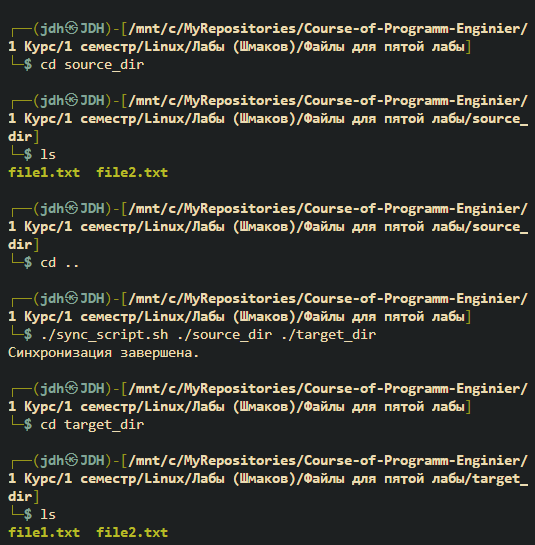
- bg переводит в фоновый режим (как запуск с &)

- jobs показывает статус процессов

4.



5.



Программа проверяет, что переданы два аргумента (исходный и целевой каталоги). Для каждого файла в целевом каталоге проверяется, существует ли соответствующий файл в исходном каталоге. Если нет, файл удаляется. Для каждого файла в исходном каталоге: если файл существует в целевом каталоге, проверяется, новее ли файл в исходном каталоге. Если да, файл обновляется. Если файл не существует в целевом каталоге, он копируется.

# ЛАБОРАТОРНАЯ РАБОТА № 6

Генерация и обработка сигналов

## Цель работы.

Освоение простейшего средства управления процессами,

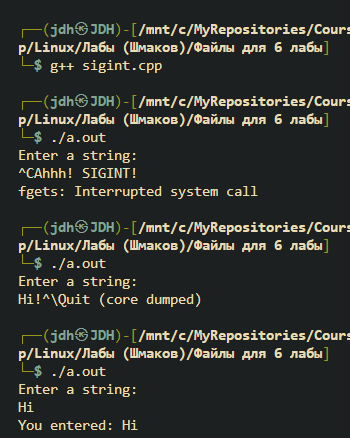
позволяющего процессам передавать информацию о каких-либо событиях,

отрабатывать реакции на различные события и взаимодействовать друг с

другом.

## Выполнение работы.

1.



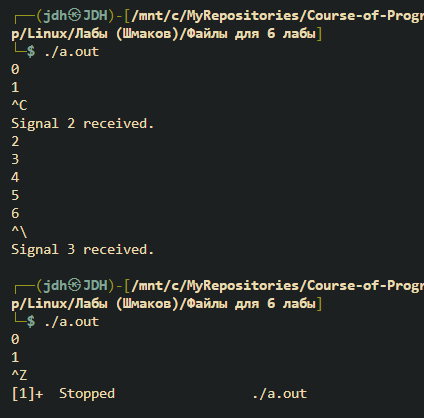
Анализ результата:

Если пользователь использует сигнал SIGINT(нажатием Ctrl-C), то выводится сообщение и завершается работа, если пользователь использует сигнал SIGQUIT (нажатием Ctrl-\), то программа просто завершает работу, а при вводе строки возвращает её в виде “You entered: {введённая строка}”.

2.

Запустите программу signal\_catch.cpp, выполняющую вывод на консоль. Отправьте процессу сигналы SIGINT и SIGQUIT , а также SIGSTOP (нажатием Ctrl-Z) и SIGCONT (нажатием Ctrl-Q) .Проанализируйте поведение процесса и вывод на консоль, а также сравните с программой из предыдущего пункта.

Выполнение:



SIGINT (Ctrl-C) и SIGQUIT (Ctrl-\) - выводят информацию о своём номере

SIGSTOP (Ctrl-Z) – останавливает программу

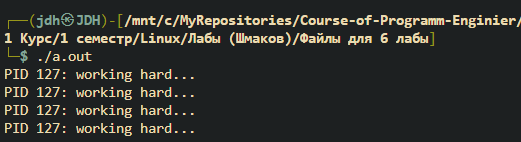
Для SIGCONT (Ctrl-Q) кода нет

3.

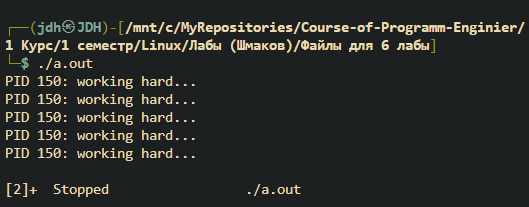
Использовал SIGQUIT



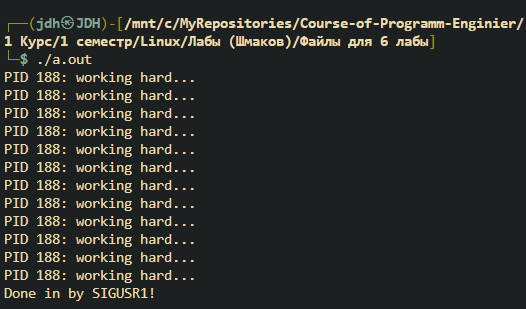
Использовал SIGINT



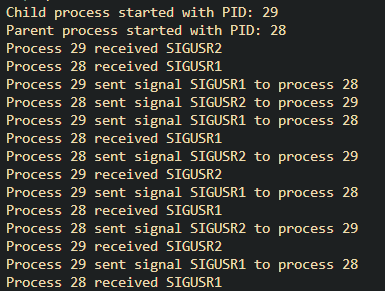
Использовал SIGSTOP



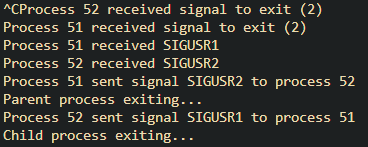
Использовал SIGCONT (продолжалось до использования SIGUSR1)



4.



5.



6.

# 

**Добавлен обработчик** SIGINT:

* + При завершении клиента по Ctrl+C удаляется private FIFO.

1. **Проверка доступности сервера**:
   * Клиент проверяет, существует ли public FIFO, и выводит сообщение об ошибке, если сервер недоступен.
2. **Удаление** private FIFO **при нормальном завершении**:
   * Клиент удаляет private FIFO при завершении, даже если это происходит не по сигналу SIGINT.

Вывод:

В ходе выполнения лабораторной работы была достигнута основная цель — освоение простейших средств управления процессами, что позволило нам глубже понять механизмы передачи информации между процессами и их взаимодействие.

# Лабораторная №7

## Цель работы

Освоение семафоров (semaphores) как эффективного

средства синхронизации доступа процессов к разделяемым ресурсам

операционной системы.

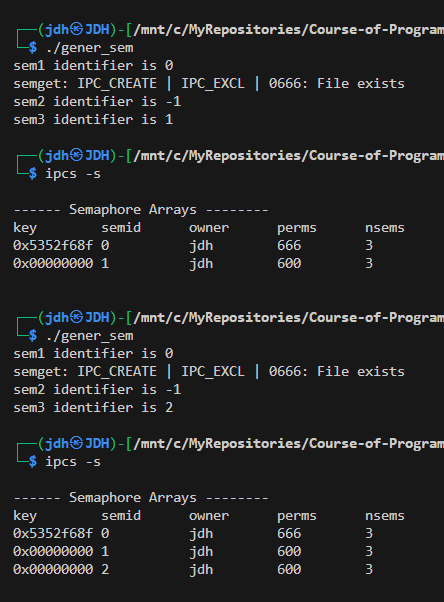
## Выполнение работы

1. Скомпилируйте и выполните программу gener\_sem.cpp,

иллюстрирующую создание наборов с семафорами или получение доступа к

ним. Запустите программу несколько раз и после каждого ее завершения

выполните команду ipcs -s

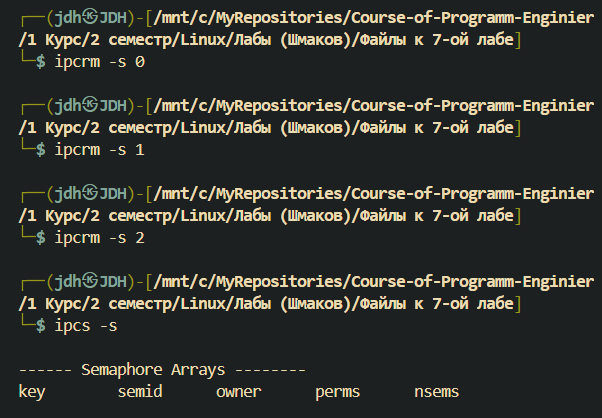


IPC\_CREAT: Позволяет создать семафор, если он не существует, или получить доступ к уже существующему.

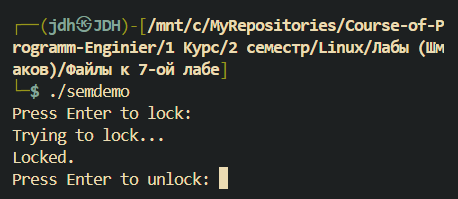
IPC\_CREAT | IPC\_EXCL: Гарантирует создание нового семафора. Если семафор уже существует, вызов завершится с ошибкой.

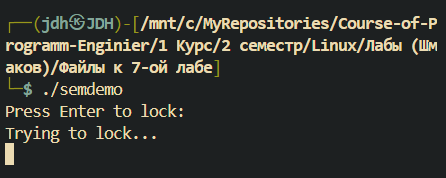
IPC\_PRIVATE: Создает уникальный семафор, который существует только во время жизни процесса.

2. Удаление созданных семафоров с помощью команды ipcrm

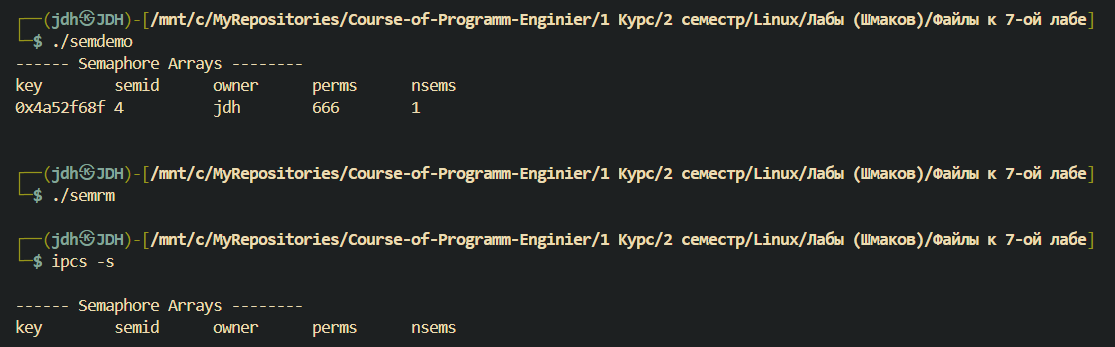


3. Программа semdemo.cpp, запущенная сразу в нескольких окнахтерминала.

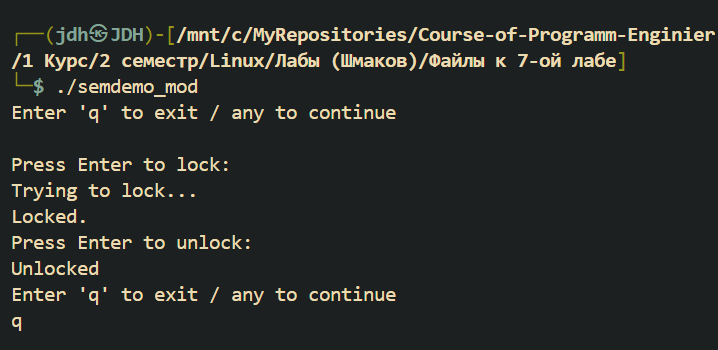




4.Удаление семафора с помощью программы semrm.cpp



5. Модификация программы semdemo.cpp



6. Программа, позволяющая мониторить количество процессов,

находящихся в состоянии ожидания освобождения ресурса в

каждый момент времени

# 

# 

Вывод:

В ходе выполнения данной работы были изучены и применены на

практике механизмы работы с семафорами в операционных

системах семейства Unix. Семафоры используются для

синхронизации доступа к общим ресурсам между процессами, что

позволяет избежать состояния гонки (race condition) и обеспечить

корректное взаимодействие процессов.

# ЛАБОРАТОРНАЯ РАБОТА № 8

Обмен через очереди сообщений

# Цель работы.

Знакомство с возможностями очередей

сообщений(Message Queues) – мощного и гибкого средства

межпроцессного взаимодействия в ОС Linux.

# Последовательность выполнения работы:

**1**. Скомпилируйте и выполните программу gener\_mq.cpp,

создающую несколько очередей сообщений. После завершения

программы выполните команду ipcs и поясните отличие результата

от того, что был при вызове подобной команды из программы



Здесь:

• key — ключ очереди сообщений.

• Msqid — идентификатор очереди.

• Owner — владелец очереди.

• Perms — права доступа (в данном случае 660, то есть rw-rw----).

• Used-bytes и messages— количество байт и сообщений в очереди

(в данном случае 0, так как очередь пуста).

После завершения программы очереди удаляются, и при повторном

запуске ipcs они не будут отображаться.

1.1 Отличие вывода ipcs в программе и в терминале

• В программе:Вывод команды ipcs отображает состояние очередей

сообщений на момент выполнения программы. Это означает, что

вы увидите все созданные программой очереди, так как они ещё не

удалены.

• В терминале: Если вы выполните команду ipcs после завершения

программы, вы не увидите созданных очередей, так как они были

удалены в конце программы с помощью msgctl(mid[i], IPC\_RMID,

NULL).

1.2 Зависимость создания очереди сообщений от флагов в msgget()

Функция msgget() используется для создания или подключения к

очереди сообщений. Её поведение зависит от переданных флагов:

Основные флаги:

• IPC\_CREAT: Создает очередь сообщений, если она не существует.

Если очередь уже существует, возвращает её идентификатор.

• IPC\_EXCL: Используется вместе с IPC\_CREAT. Если очередь уже

существует, вызов msgget() завершится с ошибкой.

• Права доступа (например, 0660): Определяют права доступа к

очереди (аналогично правам доступа к файлам в Unix).

**2**. Скомпилируйте программы sender.cpp и receiver.cpp, задав

соответствующим исполняемым файлам разные имена:( g++ <

имя .cpp файла> -o < имя .out файла > ). Запустите процессы на

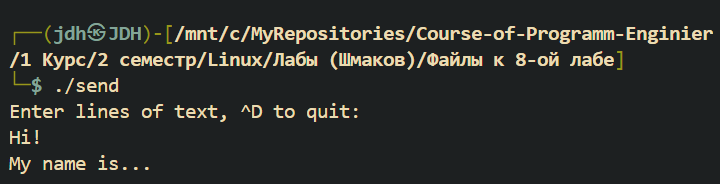
разных терминалах и передайте текстовые сообщения от процесса

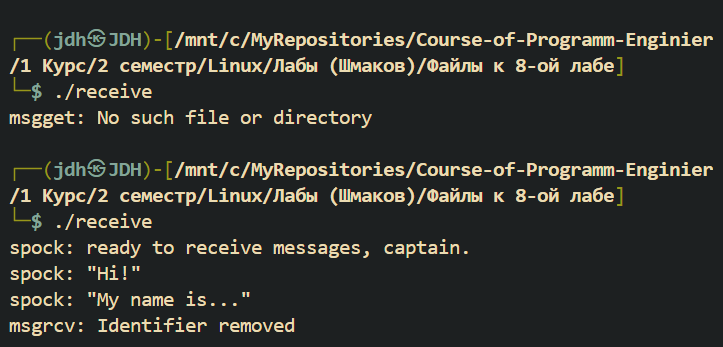
sender процессу receiver. Проанализируйте, что происходит с

ресурсом MessageQueue после завершения каждого из процессов

(командой ipcs). При этом выполните различные виды завершения

отправкой сигналов SIGQUITи SIGINT (нажатием Ctrl-C)





Анализ завершения процессов

1. Нормальное завершение (без сигналов):

• Если sender.out завершается нормально (после ввода Ctrl+D), он

удаляет очередь сообщений с помощью вызова msgctl(msqid,

IPC\_RMID, NULL).

• После завершения sender.out очередь сообщений будет удалена, и

команда ipcs -q не покажет её.

2. Завершение с помощью сигналов:

• Сигнал SIGINT (Ctrl+C):

•Если вы завершите sender.out с помощью Ctrl+C, он не

успеет удалить очередь сообщений.

•Очередь сообщений останется в системе, и её можно будет

увидеть с помощью ipcs -q.

• Сигнал SIGQUIT (Ctrl+):

•Если вы завершите sender.out с помощью Ctrl+\, он также не

удалит очередь сообщений.

•Очередь сообщений останется в системе.

3. Завершение receiver.out:

• Если вы завершите receiver.out с помощью Ctrl+C или Ctrl+\, это

не повлияет на очередь сообщений, так как receiver.out только

читает из очереди и не управляет её жизненным циклом.

• Очередь сообщений останется в системе до тех пор, пока её явно

не удалит sender.out или команда ipcrm.

Нормальное завершение sender.out: Очередь сообщений удаляется

автоматически.

1.Завершение sender.out с помощью сигналов (SIGINT или

SIGQUIT): Очередь сообщений остаётся в системе.

2. Завершение receiver.out: Не влияет на очередь сообщений, так

как receiver.out только читает из неё.

Для избежания утечек ресурсов важно всегда корректно удалять

IPC-объекты (очереди сообщений, семафоры, разделяемую память)

после их использования.

Если receiver завершился (например, с помощью Ctrl+C), а затем

запускается снова, он сможет прочитать сообщения, которые

остались в очереди после предыдущего запуска.

**3**. Что происходит, если процесс receiver запускается уже после

того,как процесс sender отправил в очередь одно или множество

сообщений, или запускается повторно после завершения?

1.Сообщения сохраняются в очереди: Если receiver запускается

после отправки сообщений, он может прочитать все сообщения,

которые были отправлены в очередь.

2.Повторный запуск receiver: Если receiver запускается повторно,

он может прочитать сообщения, которые остались в очереди после

предыдущего запуска.

3.Очередь удаляется только явно: Очередь сообщений сохраняется

до тех пор, пока её не удалит процесс (например, sender) или

администратор системы.

**4.** Запустите несколько процессов receiver на различных

терминалах и,отправляя сообщения процессом sender ,

проанализируйте ситуацию

Все процессы receiver подключены к одной и той же очереди

сообщений.

• Когда sender отправляет сообщения, они попадают в очередь.

• Сообщения из очереди могут быть прочитаны только одним из

процессов receiver. Это связано с тем, что сообщения удаляются из

очереди после их чтения.

**5.** Модифицируйте программы sender.cpp и receiver.cpp так, чтобы

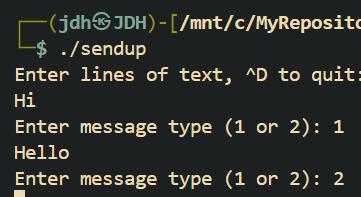
организовать отправку сообщений, по крайней мере, двух типов

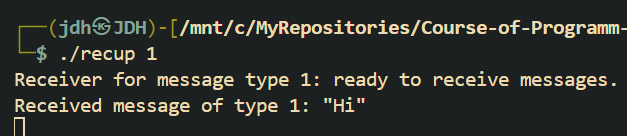
через одну и туже очередь для нескольких различных процессов

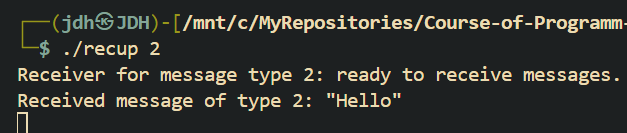
получателей. Для этого необходимо управлять параметром в поле

mtype структуры my\_msgbuf на передающей стороне и параметром

msgtyp в системном вызове msgrcv() на приемной стороне







# Лабораторная работа №9

## Цель работы

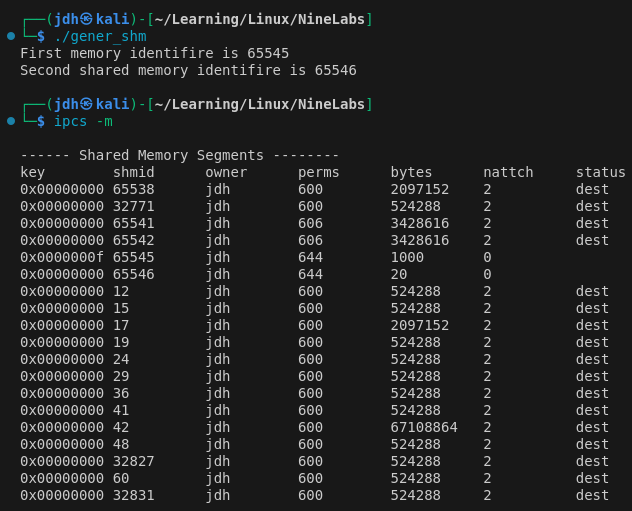
Использование для обмена данными разделяемой памяти

(shared memory) – самого быстрого средства межпроцессного взаимодействия в

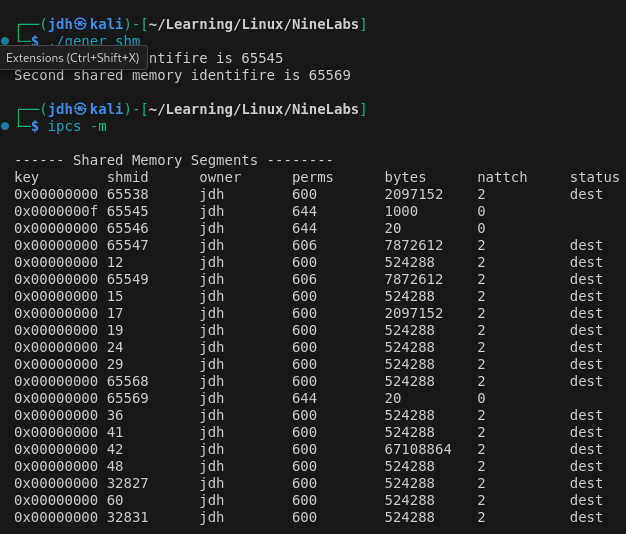
Linux.

## Последовательность выполнения работы

1.   
Первый запуск:

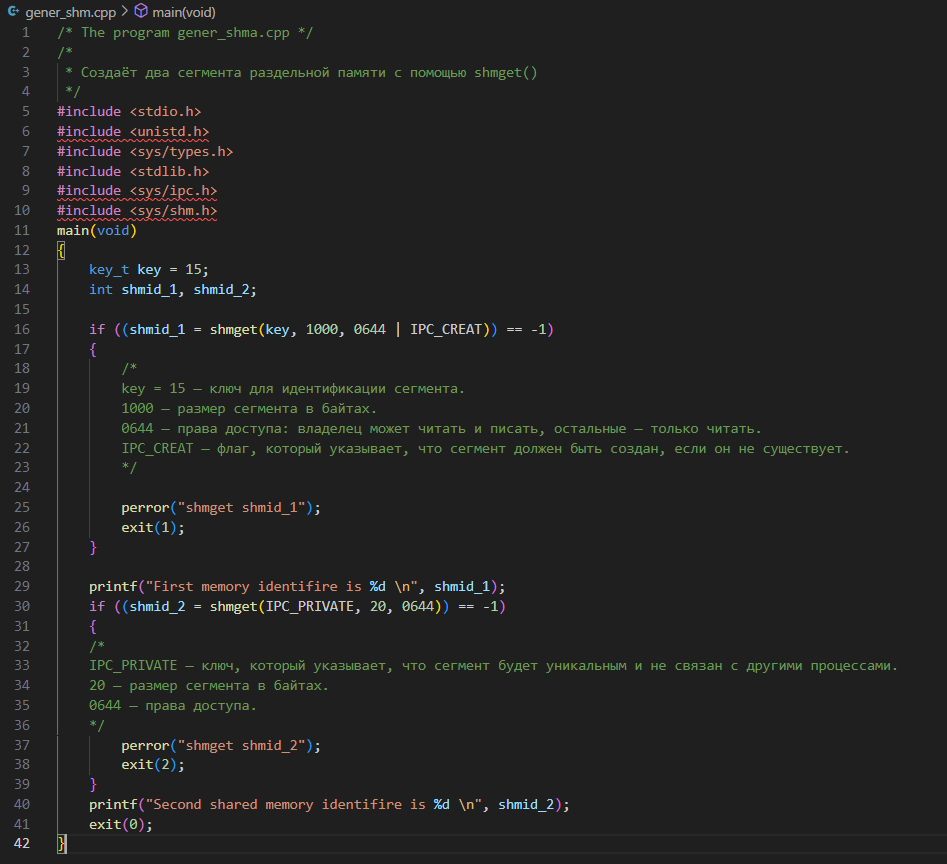


Второй запуск:

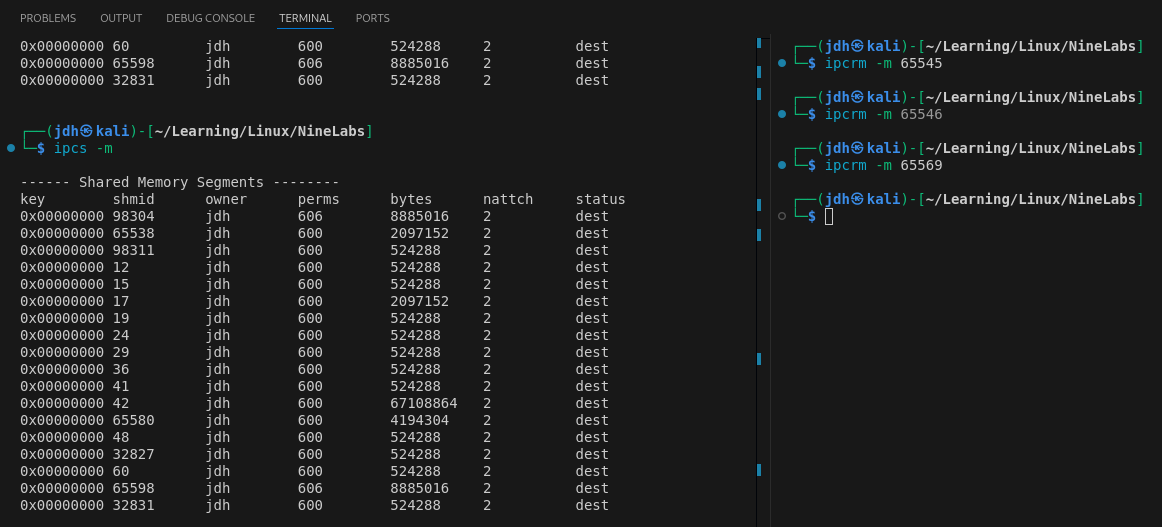


В программе создаётся два сегмента памяти. Первый — использует флаг IPC\_CREAT – такой флаг позволяет создать новый сегмент памяти по ключу, если его нет ИЛИ получить идентификатор сегмента, если он уже был создан.

IPC\_PRIVATE – КАЖДЫЙ РАЗ создаёт уникальный сегмент, который не связан с другими процессами

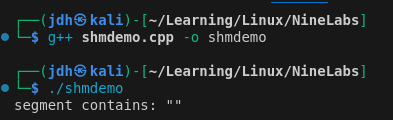


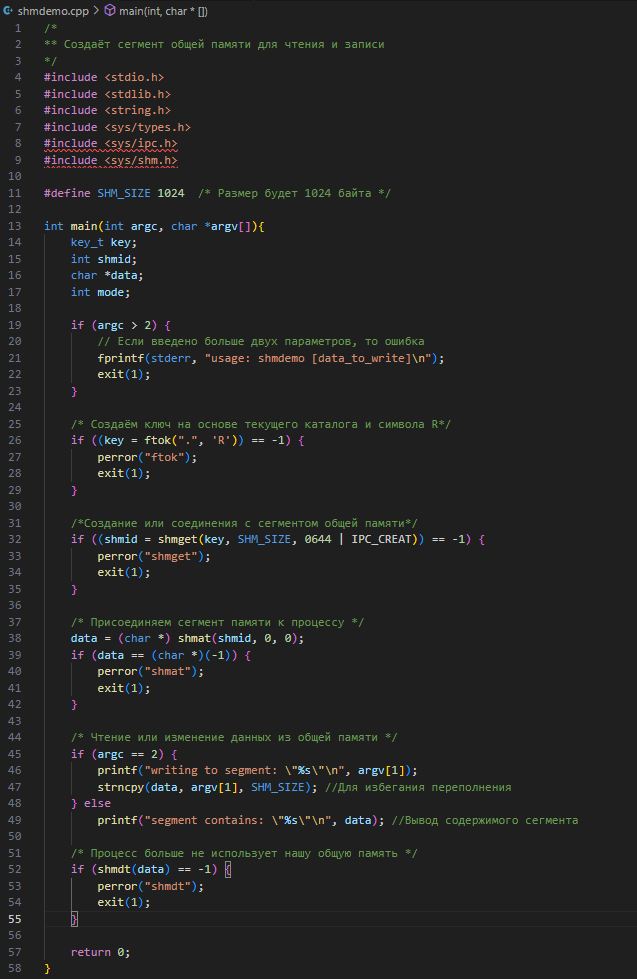
2. (Сначала правый терминал, потом левый)



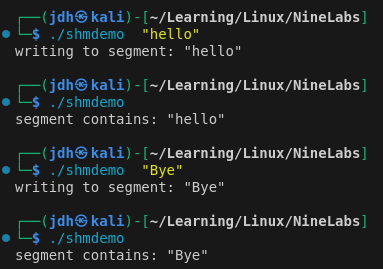
Удаление по ключу происходит аналогично

3.



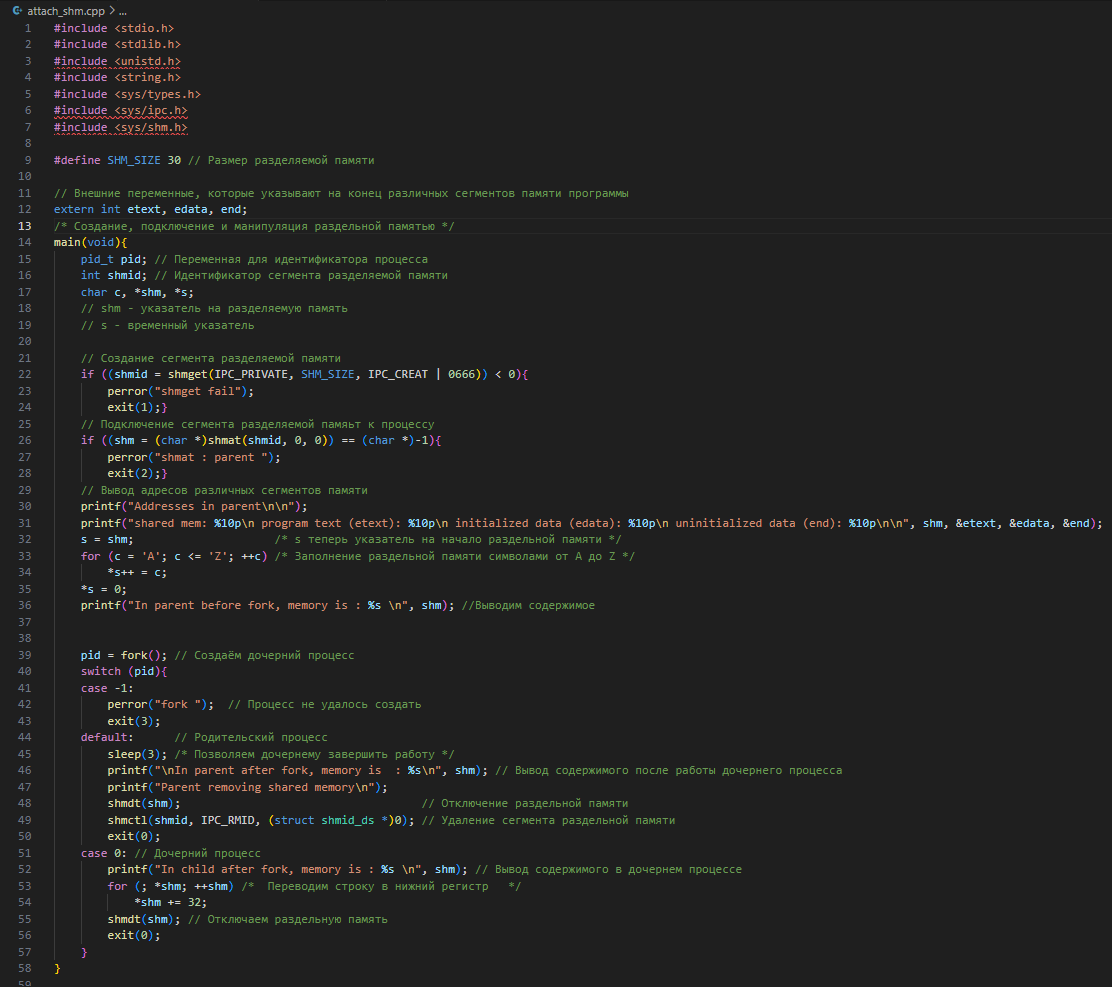


4.

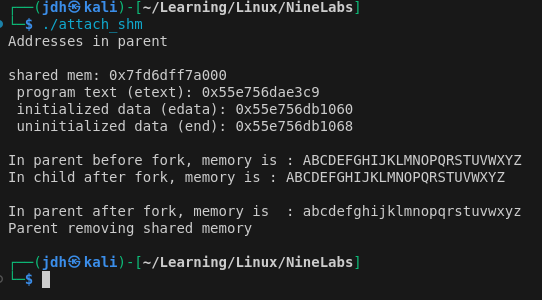


Удаление через ipcrm, как во втором шаге

5.



Запуск программы



shared mem: 0x7fd6dff7a000: Адрес, по которому разделяемая память отображается в адресном пространстве родительского процесса.

* program text (etext): 0x55e756dae3c9: Адрес конца текстового сегмента. Это место, где заканчивается исполняемый код программы.
* initialized data (edata): 0x55e756db1060: Адрес конца сегмента инициализированных данных. Здесь хранятся глобальные и статические переменные с начальными значениями.
* uninitialized data (end): 0x55e756db1068: Адрес конца сегмента неинициализированных данных (BSS). Здесь хранятся глобальные и статические переменные, не имеющие начальных значений.

**Наблюдения:**

* Адрес разделяемой памяти значительно отличается от адресов других сегментов программы (текст, данные). Это естественно, так как разделяемая память выделяется отдельно от основных сегментов программы.
* Разница между edata и end невелика (всего 8 байт). Это говорит о том, что в программе очень мало неинициализированных глобальных или статических переменных.

Модификация:

Второй параметр у shmat не 0, а адрес памяти

6. Текст:

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <string.h>

#define SHM\_SIZE 1023  // Размер каждого сегмента разделяемой памяти

int main() {

    int shmid1, shmid2, shmid3;  // Идентификаторы сегментов разделяемой памяти

    char \*shm1, \*shm2, \*shm3;    // Указатели на разделяемую память

    // Создание первого сегмента разделяемой памяти

    if ((shmid1 = shmget(IPC\_PRIVATE, SHM\_SIZE, IPC\_CREAT | 0666)) < 0) {

        perror("shmget fail for shmid1");

        exit(1);

    }

    // Создание второго сегмента разделяемой памяти

    if ((shmid2 = shmget(IPC\_PRIVATE, SHM\_SIZE, IPC\_CREAT | 0666)) < 0) {

        perror("shmget fail for shmid2");

        exit(1);

    }

    // Создание третьего сегмента разделяемой памяти

    if ((shmid3 = shmget(IPC\_PRIVATE, SHM\_SIZE, IPC\_CREAT | 0666)) < 0) {

        perror("shmget fail for shmid3");

        exit(1);

    }

    // Подключение первого сегмента к адресному пространству процесса

    if ((shm1 = (char \*)shmat(shmid1, 0, 0)) == (char \*)-1) {

        perror("shmat fail for shm1");

        exit(2);

    }

    // Подключение второго сегмента к адресному пространству процесса

    if ((shm2 = (char \*)shmat(shmid2, 0, 0)) == (char \*)-1) {

        perror("shmat fail for shm2");

        exit(2);

    }

    // Подключение третьего сегмента к адресному пространству процесса

    if ((shm3 = (char \*)shmat(shmid3, 0, 0)) == (char \*)-1) {

        perror("shmat fail for shm3");

        exit(2);

    }

    // Вывод адресов, по которым были подключены сегменты

    printf("Segment 1 attached at address: %p\n", shm1);

    printf("Segment 2 attached at address: %p\n", shm2);

    printf("Segment 3 attached at address: %p\n", shm3);

    // Проверка, размещены ли сегменты в последовательных участках памяти

    if (shm2 == shm1 + SHM\_SIZE && shm3 == shm2 + SHM\_SIZE) {

        printf("Segments are placed consecutively in memory.\n");

    } else {

        printf("Segments are NOT placed consecutively in memory.\n");

    }

    // Попытка доступа к 1024-му байту каждого сегмента

    printf("\nAttempting to access the 1024th byte of each segment:\n");

    // Попытка доступа к 1024-му байту первого сегмента

    printf("Accessing 1024th byte of segment 1: ");

    shm1[1023] = 'A';  // 1023-й байт (последний допустимый)

    printf("Success (last valid byte).\n");

    printf("Accessing 1024th byte of segment 1: ");

    shm1[1024] = 'B';  // 1024-й байт (выход за пределы)

    printf("This should cause a segmentation fault.\n");

    // Отключение сегментов от адресного пространства процесса

    shmdt(shm1);

    shmdt(shm2);

    shmdt(shm3);

    // Удаление сегментов разделяемой памяти

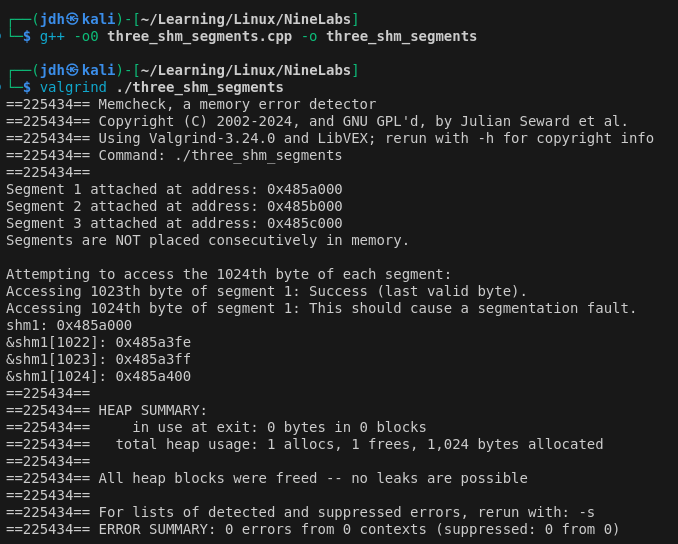
    shmctl(shmid1, IPC\_RMID, NULL);

    shmctl(shmid2, IPC\_RMID, NULL);

    shmctl(shmid3, IPC\_RMID, NULL);

    return 0;

}



Как видно, они разделены 4096 байтами. Есть предположение, что моя система выравнивает память именно так для более эффективного использования памятью. Обращение к shm1[4096] подтвердило предположение, но на такое не стоит рассчитывать не стоит

7.

Я запустил parrent.cpp со значениями 5 5 Вот, что вывело:

P: [0] One polka\_dot tree ran out.

C: [0] One polka\_dot tree ran out.

P: [1] A yellow spider broke out.

C: [1] A yellow spider broke out.

P: [2] The reddump truck fell out.

C: [2] The reddump truck fell out.

P: [3] One reddump truck ran away.

C: [3] One reddump truck ran away.

P: [4] The yellow spider broke away.

C: [4] The yellow spider broke away.

P: [5] A reddump truck fell out.

C: [5] A reddump truck fell out.

P: [0] One yellow spider broke down.

C: [0] One yellow spider broke down.

P: [1] One polka\_dotdump truck ran away.

C: [1] One polka\_dotdump truck ran away.

P: [2] A yellowdump truck broke down.

P: [3] One reddump truck fell down.

C: [2] A yellowdump truck broke down.

P: [4] A polka\_dotdump truck fell away.

C: [3] One reddump truck fell down.

P: [5] A yellowdump truck fell out.

C: [4] A polka\_dotdump truck fell away.

C: [5] A yellowdump truck fell out.

P: [0] A yellowdump truck fell down.

C: [0] A yellowdump truck fell down.

P: [1] The yellowdump truck fell down.

P: [2] A yellow spider ran out.

C: [1] The yellowdump truck fell down.

semop -- consumer -- acquire : Invalid argument

**Ключевые моменты:**

* **Разделяемая память:** Используется для обмена данными между producer и consumer.
* **Семафоры:** Используются для синхронизации доступа к разделяемой памяти и предотвращения гонок данных.
* **Producer:** Генерирует данные и записывает их в разделяемую память.
* **Consumer:** Читает данные из разделяемой памяти и отображает их.
* **Родительский процесс:** Создает разделяемую память, семафоры и запускает producer и consumer.

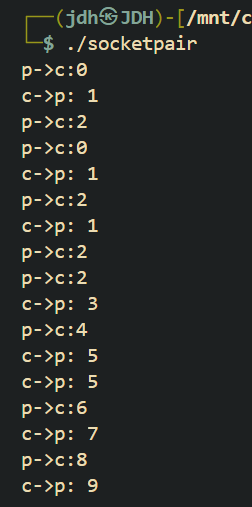
# Лабораторная работа №10

## Цель работы

Освоение набора системных вызовов для создания сокетных соединений различных типов для обмена данными по сети.

## Последовательность выполнения работы

**1.** Скомпилируйте и выполните программу socketpair.cpp , иллюстрирующую создание сокета упрощенного вида и обмен данными двух родственных процессов. Проанализируйте вывод на консоль. Существует ли зависимость обмена от различных соотношений величин временных задержек (в вызовах sleep()) в процессе родителе и в процессе потомке?



p->c:0 // Родитель получил 0 от потомка

c->p: 1 // Потомок получил 1 от родителя

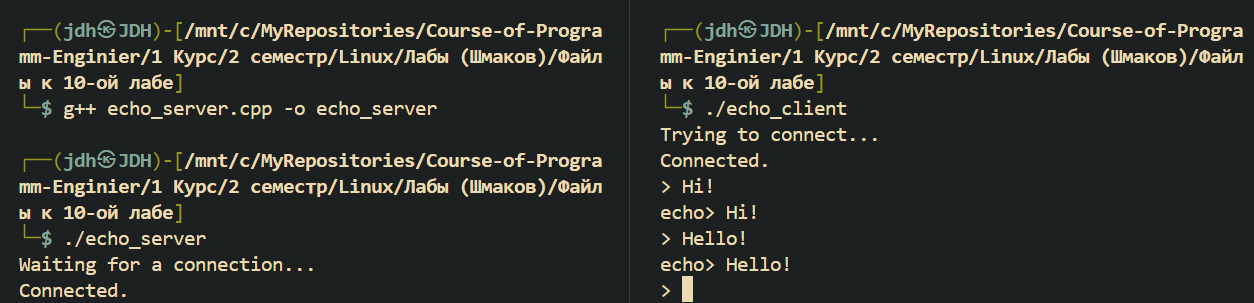
Программа создает пару сокетов с помощью socketpair().

• Родительский и дочерний процессы обмениваются

сообщениями через эти сокеты.

От sleep ничего не зависит, так как есть read и write (они блокируют выполнение пока сообщение не будет прочитано)

**2.** Скомпилируйте программы echo\_server.cpp и echo\_client.cpp , задавая им при компиляции разные имена (размещаем файлы в одном каталоге). Запустите программы сервера и клиента на разных терминалах. Вводите символьную информацию в окне клиента и проанализируйте вывод. Какому типу принадлежат сокеты, используемые в данном примере клиент- серверного взаимодействия? С чем связано создание специального файла в текущем каталоге во время исполнения программ?

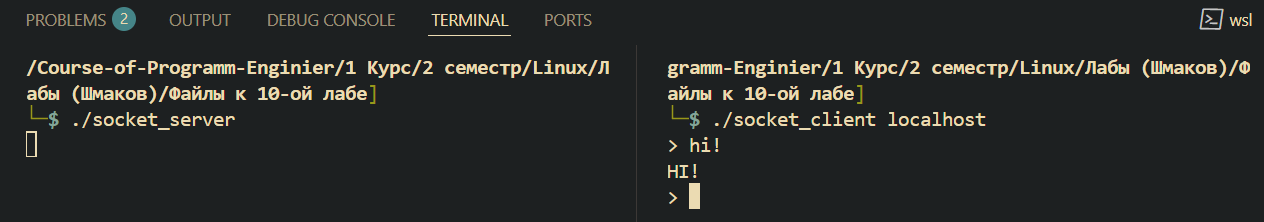
****

В данном примере используются доменные сокеты (обычно с типом SOCK\_STREAM для TCP-соединения). Они обеспечивают надежное, направленное соединение между клиентом и сервером.

Создание специального файла в текущем каталоге может использоваться для работы с сокетами Unix Domain. Обычно, сервер создает файл сокета, чтобы клиенты могли к нему подключаться через механизм файловой системы.

**3.** Скомпилируйте c разными именами программы sock\_c\_i\_srv.cpp и sock\_c\_i\_clt.cpp (в них используется общий include файл local\_c\_i.h).

Запустите программы сервера и клиента на разных терминалах. При запуске клиента указывайте в качестве параметра командной строки имя хоста localhost. Вводите символьную информацию в окне клиента и поясните вывод. Какому типу принадлежат сокеты, используемые в данном примере клиент- серверного взаимодействия?



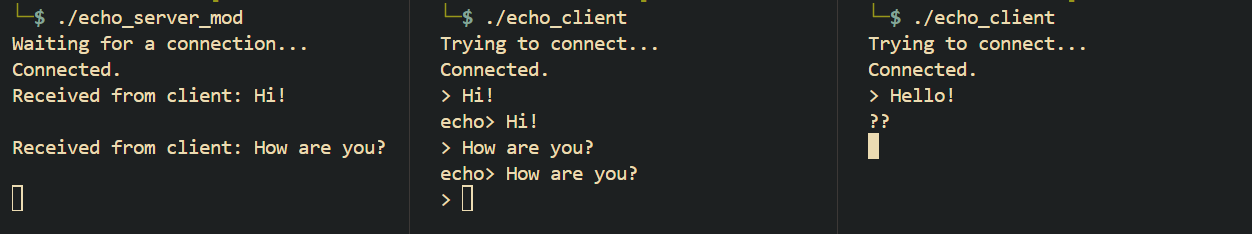
В данном примере используются сокеты типа SOCK\_STREAM, что

соответствует TCP-сокетам. Это означает, что используется

надежное, ориентированное на соединение взаимодействие между

клиентом и сервером.

**4.** Модифицируйте программу echo\_server.cpp так, чтобы при ответе на запросы клиента что-либо выводилось в окне сервера. Испытайте работу эхо-сервера при одновременной работе с несколькими клиентами.



После отключения первого клиента, все сообщения, которые отправил второй автоматически выведутся на сервер.

Вывод: В ходе выполнения данной работы были реализованы и протестированы два типа клиент-серверного взаимодействия: с использованием интернет-сокетов (TCP) и UNIX-сокетов. Каждый из подходов имеет свои особенности и области применения, что было продемонстрировано в ходе выполнения задания.

# ЛАБОРАТОРНАЯ РАБОТА №11

## Цель работы

Создание клиент-серверных приложений, взаимодей-

ствующих друг с другом по сети на основе технологии соединения

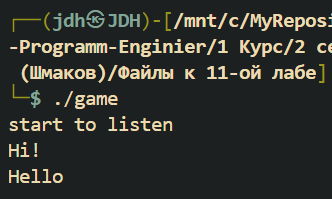
на сокетах.

## Последовательность выполнения работы

**1.** Скомпилируйте и запустите программу server\_game.cpp ,

иллюстрирующую обмен данными с клиентскими приложениями по

итеративной схеме.

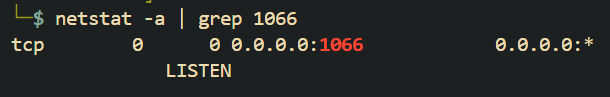


**2**. Запустите другой терминал и проверьте с него наличие в системе

созданного сервером сокета и то, что он находится в состоянии LISTEN. Для

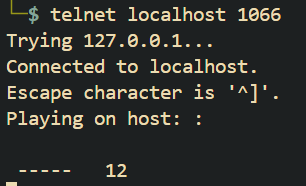
этого выполните команду netstat -a | grep 1066 . Проанализируйте вывод

данной команды и объясните ее смысл.

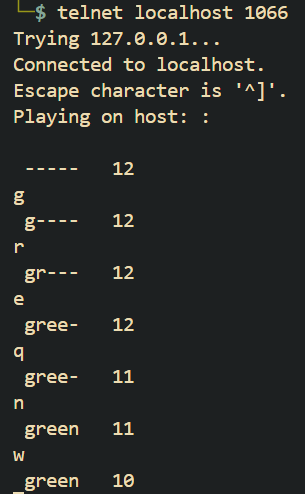


Это подтверждает, что сервер создал сокет и находится в состоянии LISTEN, ожидая подключений на порту 1066.

**3.** Запустите в качестве клиентского процесса утилиту telnet localhost 1066 . При организации коммуникации по сети, на разных компьютерах вместо localhost при запуске клиента указывается IP-адрес компьютера, на котором был запущен сервер.

****

**4.** Диалог с сервером заключается в угадывании слова. Оно вводится по буквам с клиентского терминала. При этом сервер вместо неугаданных букв выдает символы ”-” , а также считает число оставшихся неудачных попыток (всего их предусмотрено 12).

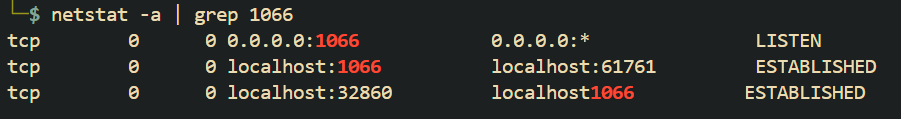
****

**5.** Завершите серверное приложение с помощью сигнала kill , и затем

определите командой netstat -a | grep 1066 когда исчезает из системы

соединение на сокетах. Во время сеанса обмена также примените команду

netstat -a | grep 1066 , чтобы исследовать состояние соединения.

****

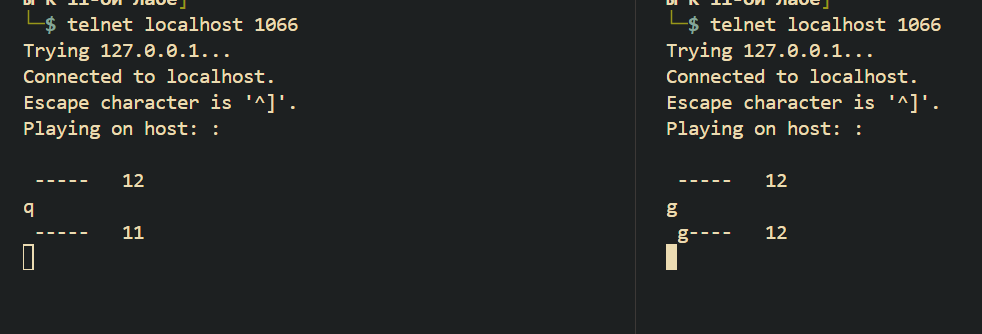
**6.** Проделайте все заново, но запускайте не одно клиентское приложение (в виде telnet), а несколько экземпляров с разных терминалов, и попытайтесь работать с них одновременно. Проанализируйте, как сервер будет обслуживать запросы в этом случае.

Когда несколько клиентов одновременно пытаются подключиться, сервер принимает первое подключение

Все остальные клиенты "зависают" в состоянии ожидания

Только после завершения игры с первым клиентом (либо угадывания слова, либо исчерпания попыток) сервер переходит к обработке следующего подключения

**7.** Модифицируйте программу server\_game.cpp так, чтобы запросы от каждого из клиентов могли обслуживаться конкурентно (путем запуска для каждого нового соединения собственного нового процесса на сервере. Возможно также улучшить качество самой игровой функции guess\_word() сервера. Проанализируйте, как обслуживаются запросы в случае конкурентной схемы работы сервера.

При каждом новом соединении создаётся отдельный дочерний процесс, который обслуживает клиента независимо от других