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: 28](#__RefHeading___Toc2225_714890939)

[ЛАБОРАТОРНАЯ РАБОТА №4 29](#__RefHeading___Toc913_1721718268)

[Цель работы. 29](#__RefHeading___Toc6341_1721718268)

[Выполнение работы 29](#__RefHeading___Toc1376_2152403033)

[Вывод. 31](#__RefHeading___Toc6339_1721718268)

[Лабораторная работа №5 32](#__RefHeading___Toc3778_1517724926)

[Цель работы 32](#__RefHeading___Toc3780_1517724926)

[Лабораторная №7 32](#__RefHeading___Toc3782_1517724926)

[Цель работы 32](#__RefHeading___Toc3784_1517724926)

[Выполнение работы 32](#__RefHeading___Toc3786_1517724926)

[Лабораторная работа №9 34](#__RefHeading___Toc1378_2152403033)

[Цель работы 34](#__RefHeading___Toc1380_2152403033)

[Последовательность выполнения работы 34](#__RefHeading___Toc1382_2152403033)

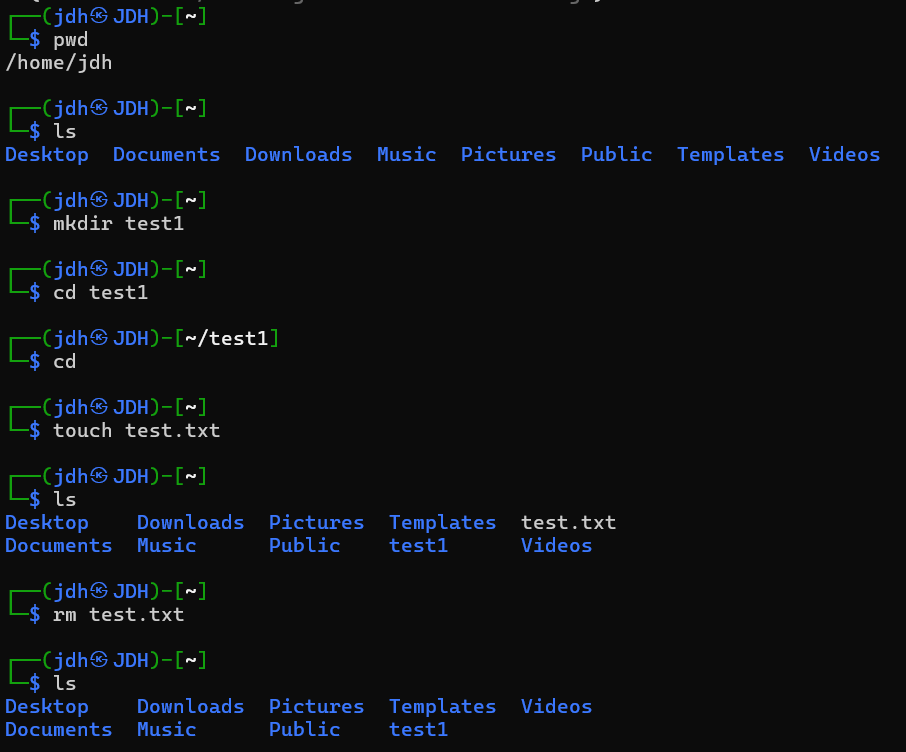
# **Лабораторная работа №1**

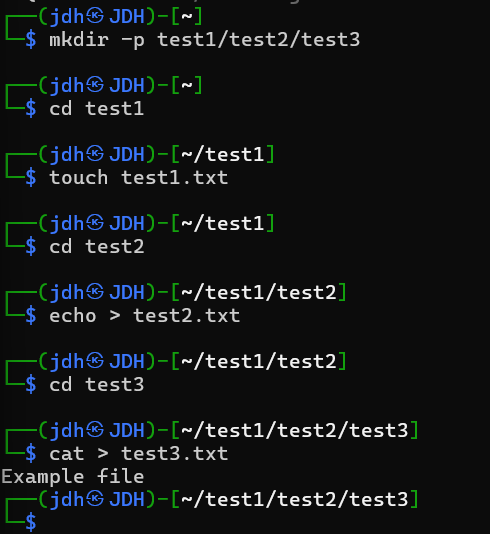
**Базовые команды ОС Linux**

## **Цель работы**

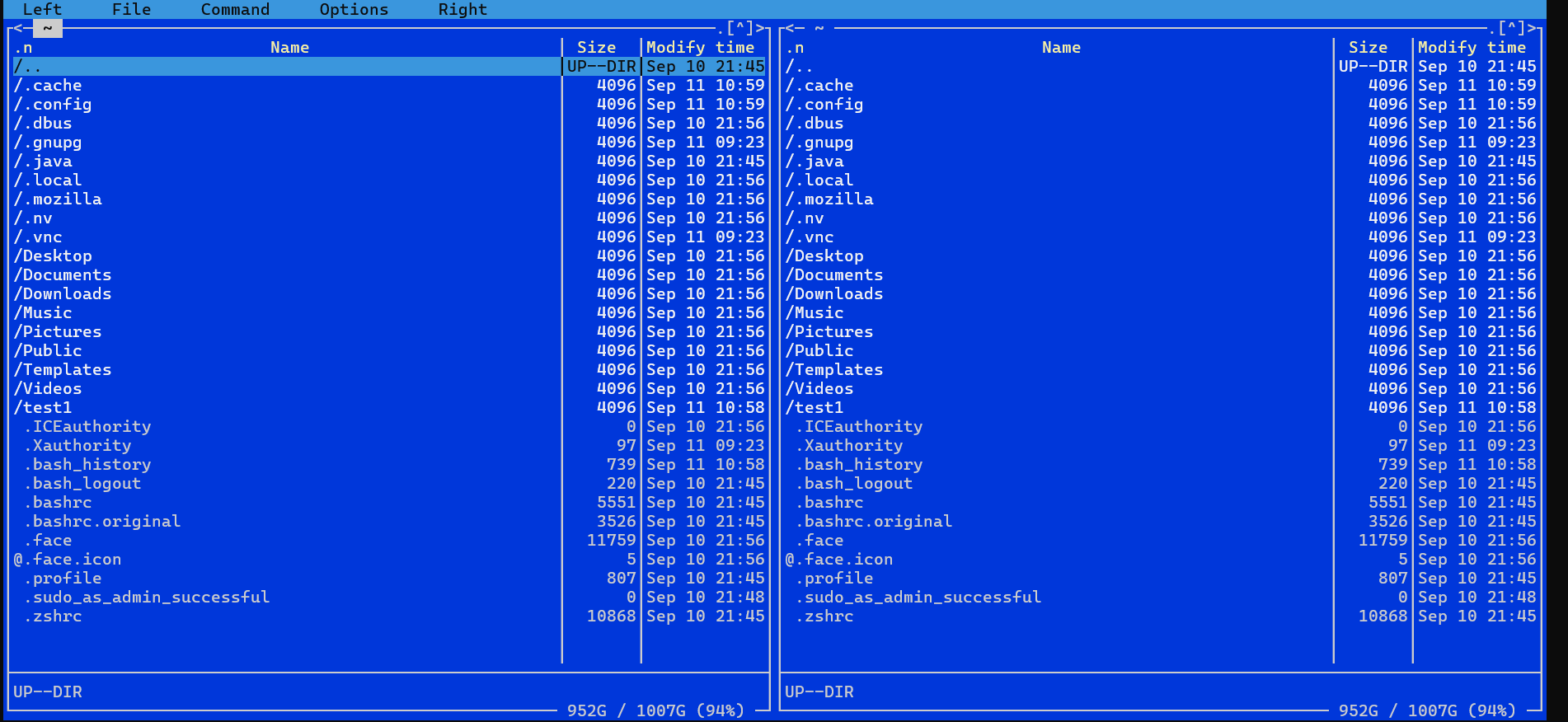
Освоение минимального набора базовых команд операционной системы Linux, знакомство с файловой системой, особенностями прав доступа, получение первичных навыков работы под Linux.

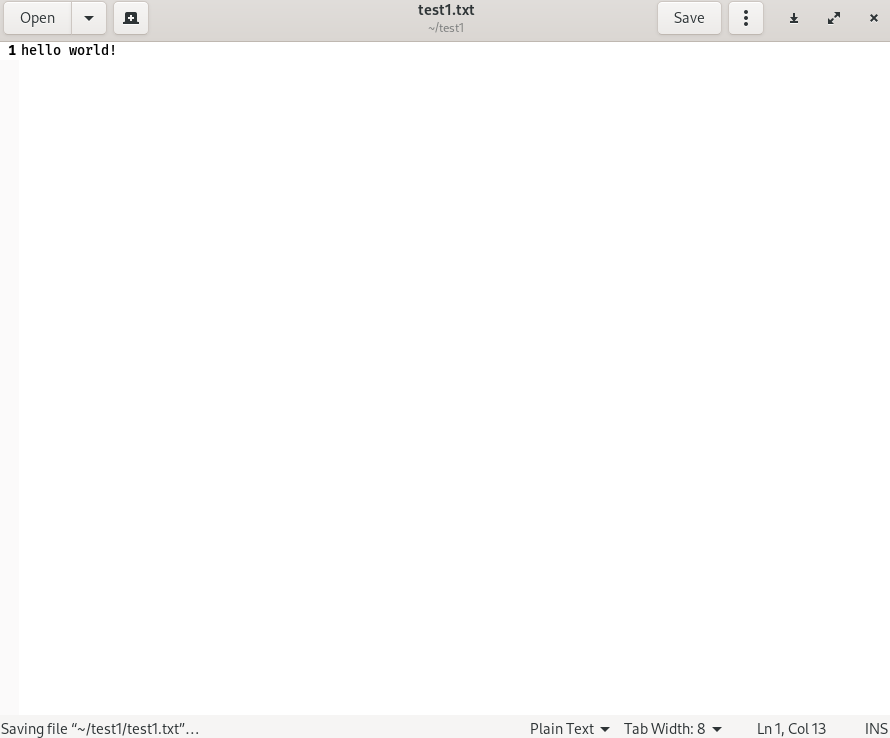
## **Последовательность выполнения работы:**

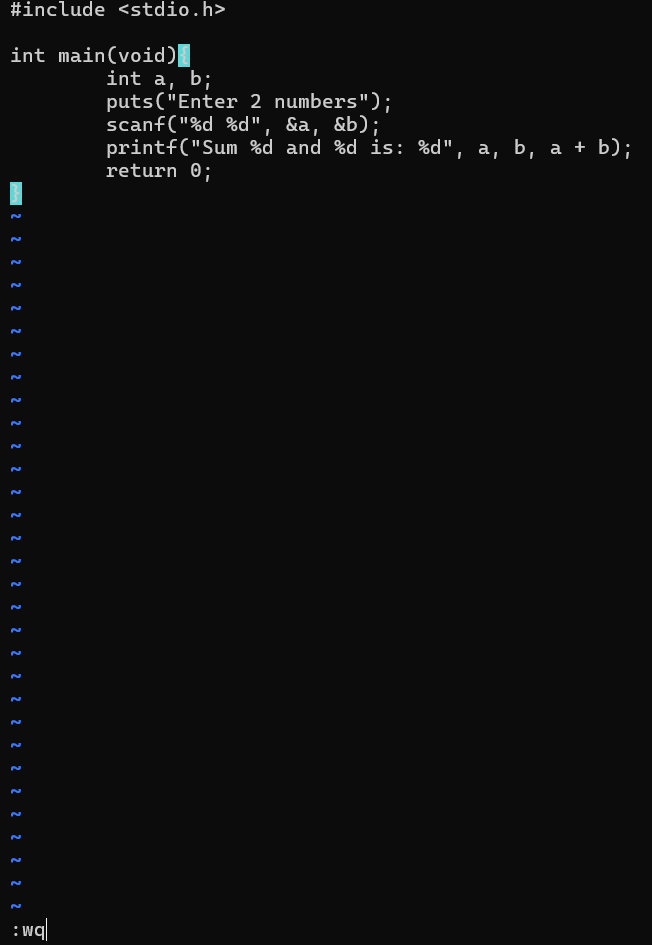
1. Запустил терминал и выполнил серию команд shell, такие, как pwd , ls , cd , mkdir , rm
2. 



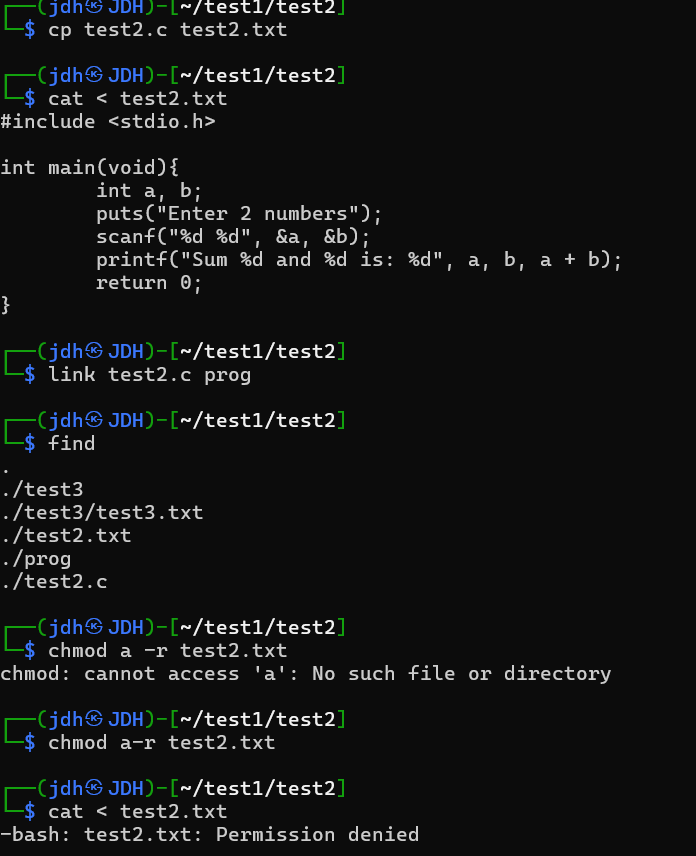
4.

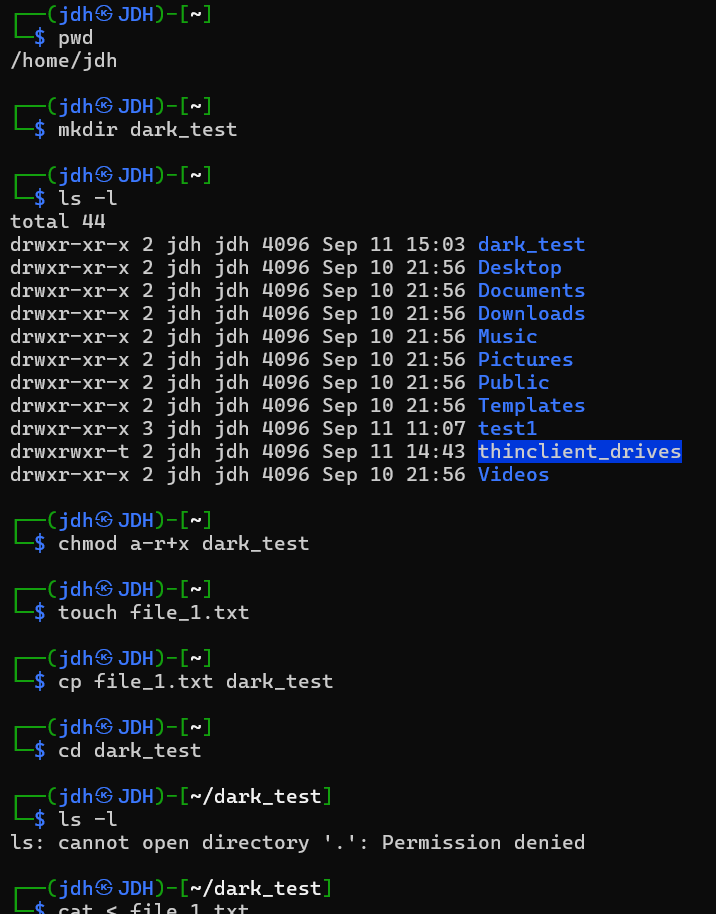






5.



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

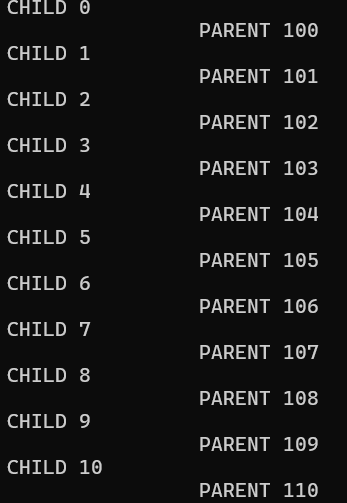
## Цель работы

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**

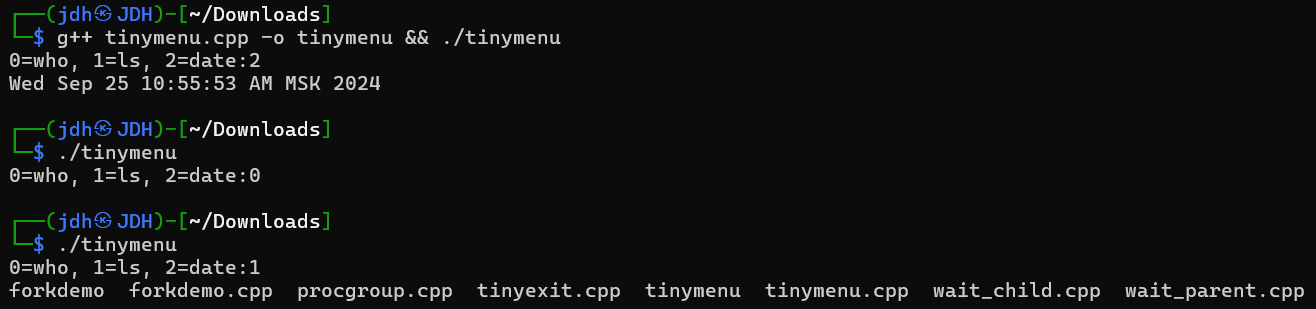
1. Запуск и компиляция файлов:

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

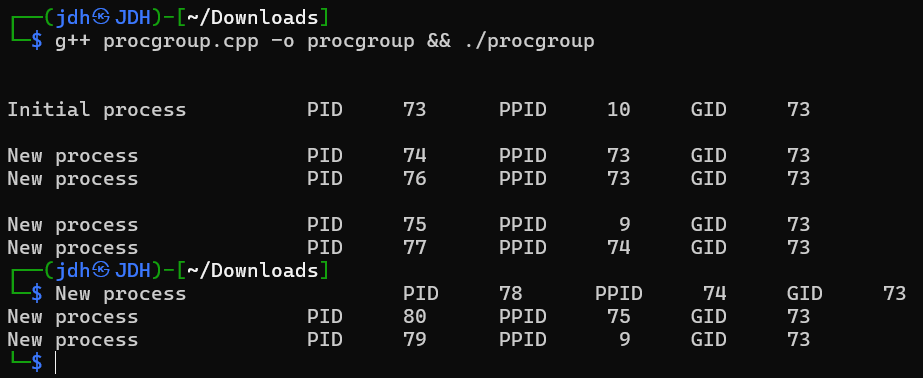


Функция execlp вызывает выбранную функцию из массива cmd.

3) tinyexit.cpp

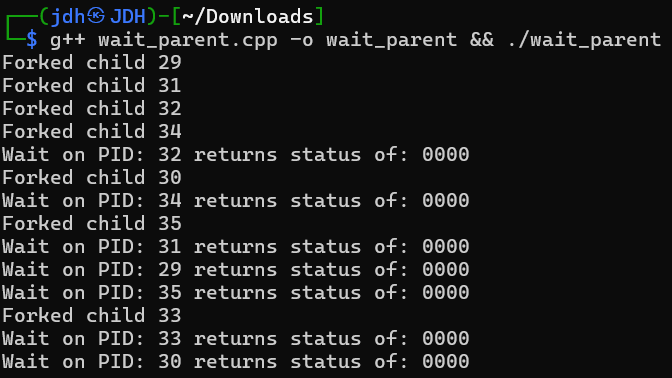
Программа (так же, как и предыдущая) выполняет указанную команду, но не завершается после каждой выполненной команды.

4) procgroup.cpp



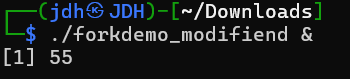
Программа генерирует копии процессов.

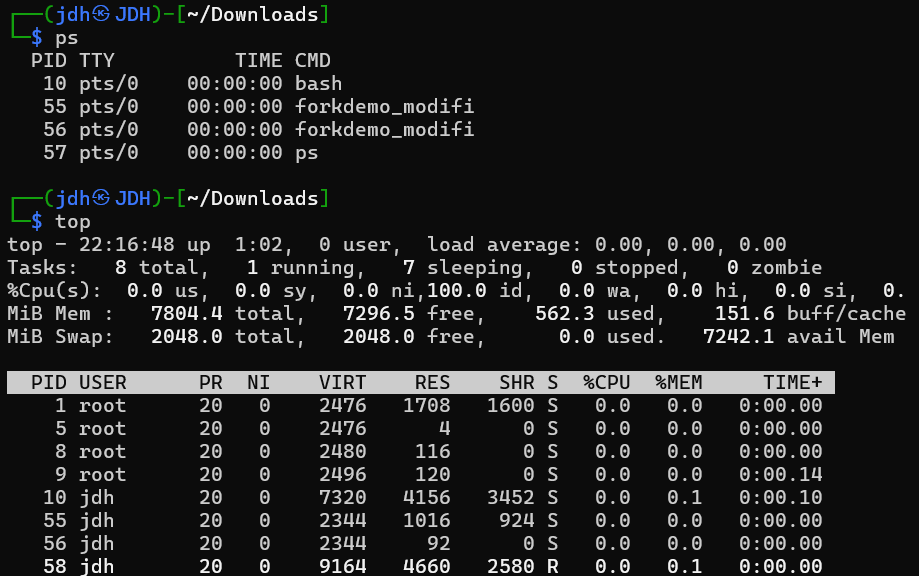
5) wait\_parent.cpp

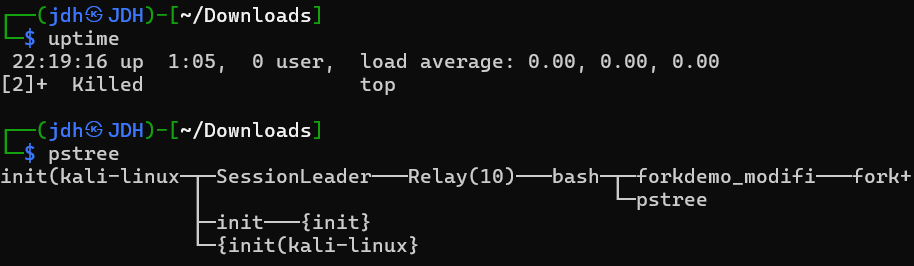


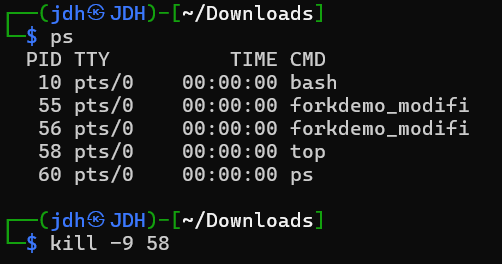
Процесс wait\_parent запускает процесс wait\_child, а затем ожидает завершения своих дочерних процессов.

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:

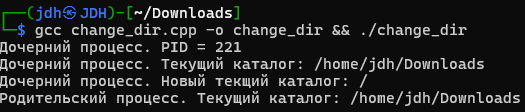
1. ps позволяет получить информацию о процессах и работать с ними
2. top выводит список работающих в системе процессов и информацию о них в отсортированном по нагрузке на процессор виде (причём в реальном времени).
3. uptime даёт информацию о: текущем времени, онлайн-пользователях, длительности работы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

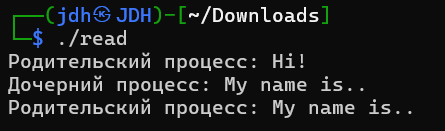
Если процесс-потомок сменит текущий каталог, текущий каталог для родителя изменяться не будет. Подтвердим предположение с помощью специально разработанного кода.

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

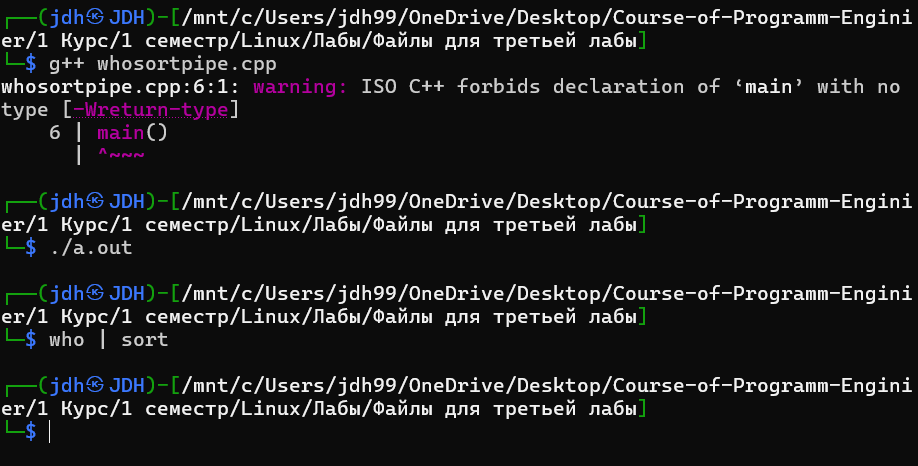
# ЛАБОРАТОРНАЯ РАБОТА №3

## Цель работы:

Знакомство с программными каналами

Программные каналы

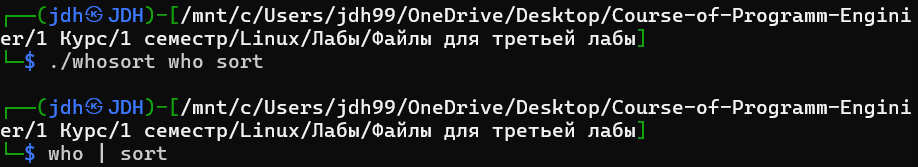
1. Скомпилируйте и выполните программу whosortpipe.cpp. Сопоставьте результат выполнения программы с выполнением этих же двух команд с терминала в конвейерном режиме (с использованием <>). Анализ результатов работы этой программы (как и всех последующих) с соответствующими скриншотами приведите в отчете.



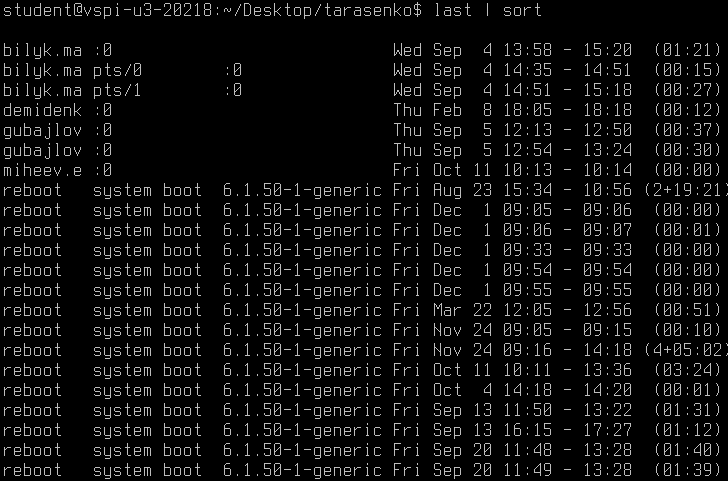
Результаты абсолютно одинаковы.

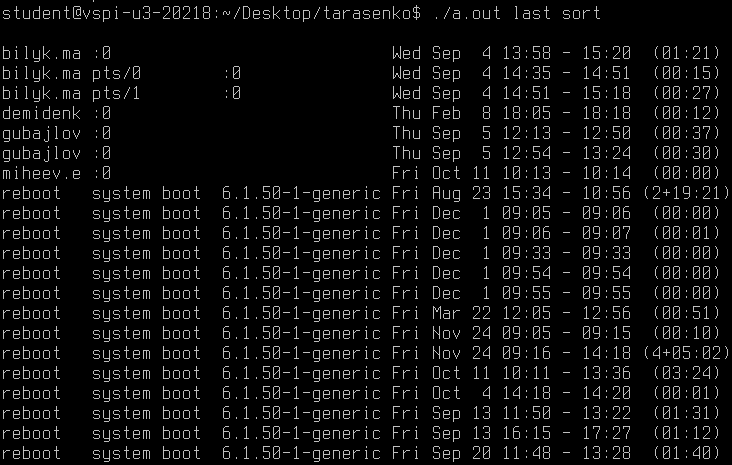
2. Программу cmdpipe.cpp запускайте после компиляции, задавая ей при стартах в качестве параметров командной строки пары команд shell для конвейеризации (who и sort ; last и sort ; last и more ; pstree и more). Сопоставьте результаты запусков программы с выполнением тех же пар командиз shell в конвейерном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

2.2 Сравнение запуска с who sort:

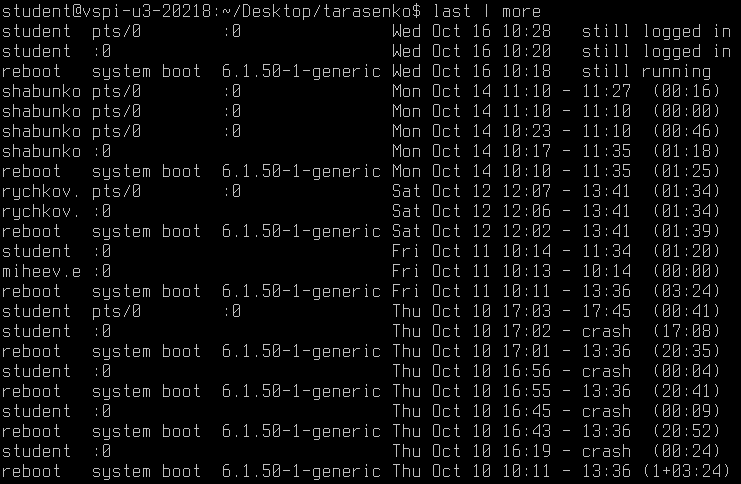


2.3 Сравнение запуска с last sort:





2.4 Сравнение запуска с last more:



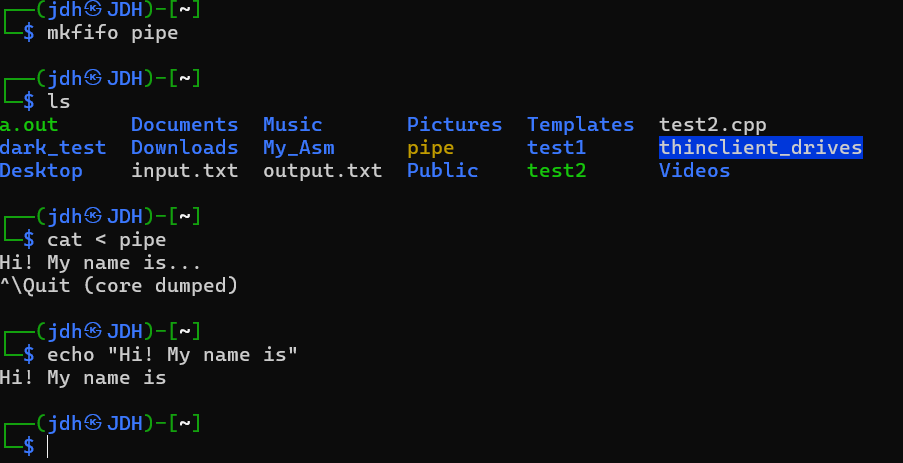
2.5 Сравнение запуска с pstree more:



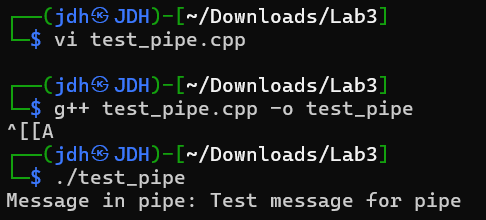


3. Создайте программный канал типа named pipe из командной строки (пример в лекциях). Используя необходимые системные вызовы организуйте канал named pipe в программе, сравните результат выполнения обмена по нему в программе с тем, чего можно достичь, создавая named pipe на терминале.

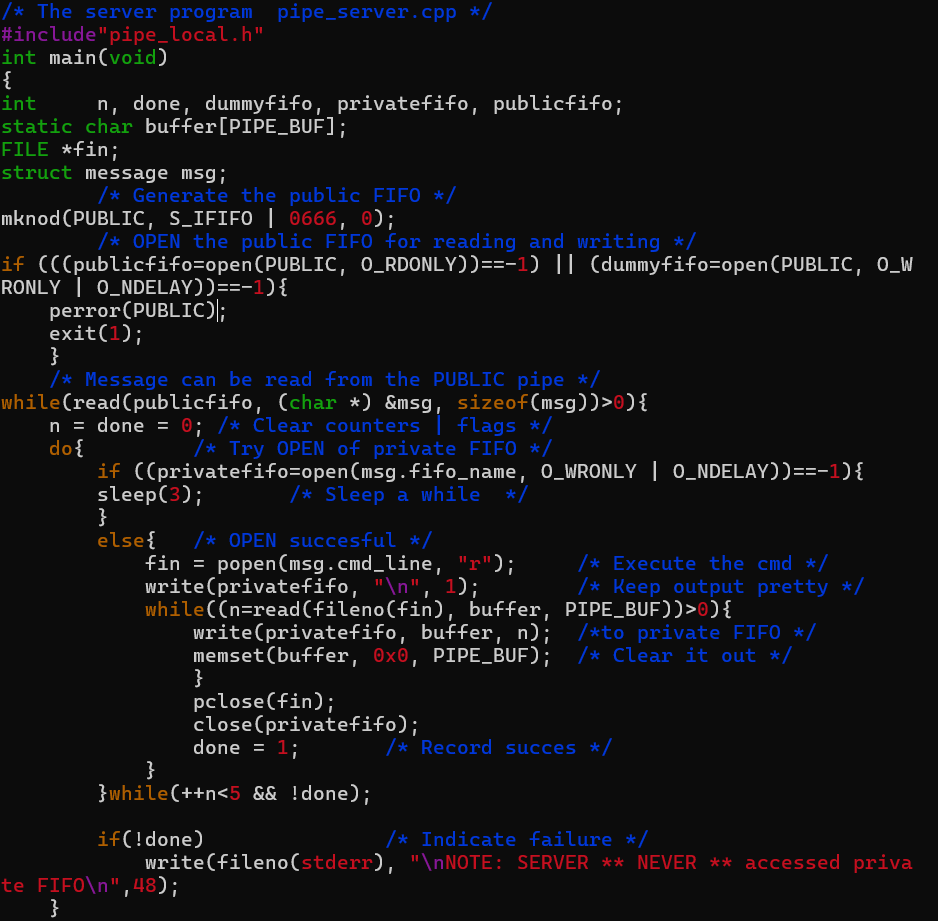
3.1 shell



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.





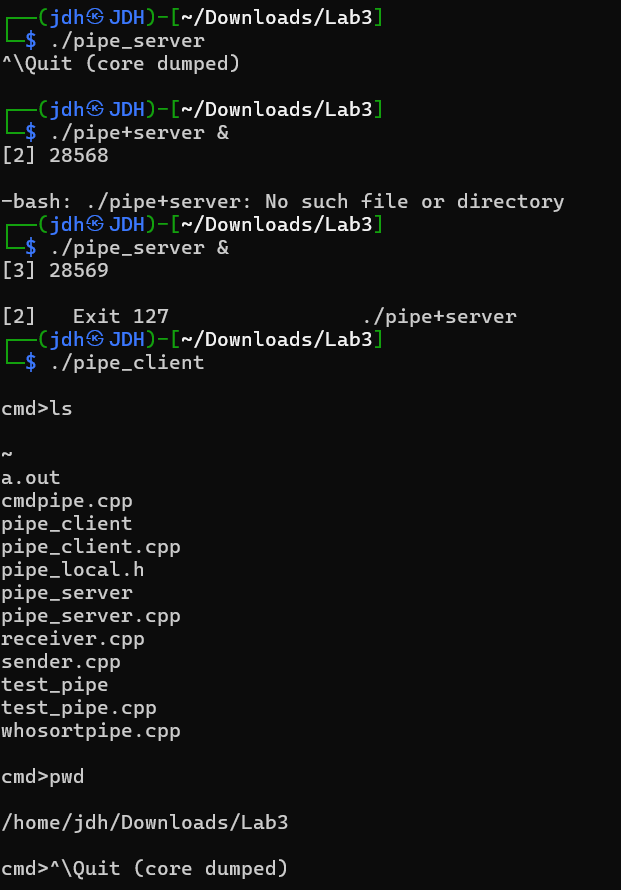
******

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.

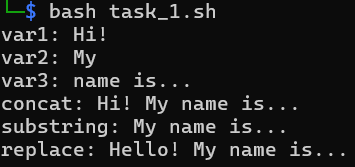
# ЛАБОРАТОРНАЯ РАБОТА №4

## Цель работы.

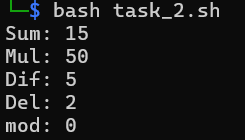
Знакомство с важным атрибутом любой операционной системы – переменными среды (или переменными окружения) environment variables и с возможностями их использования в Linux. Освоение языка для составления командных сценариев и написание набора полезных для системного администрирования скриптов.

## Выполнение работы

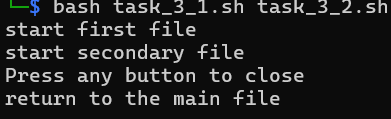
1. Создаем переменные в bash файле, выполняет конкатенацию строк, извлечение подстроки и заменяем значение в конкатенированной строке на произвольное



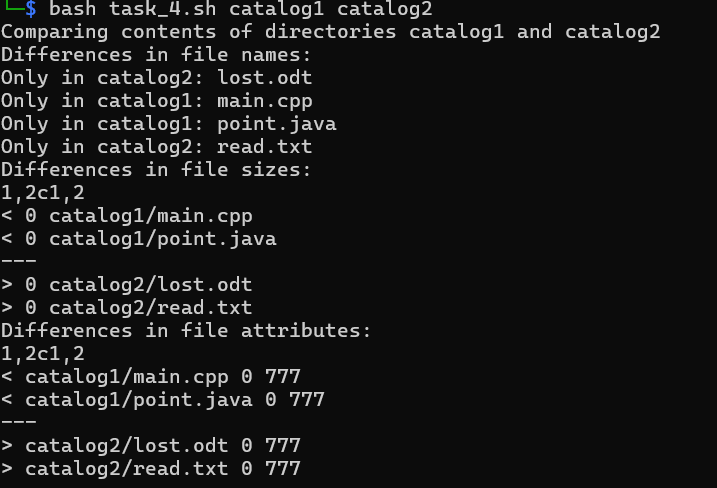
2. Создаем числовые переменные и выполняем с ними арифметические операции



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

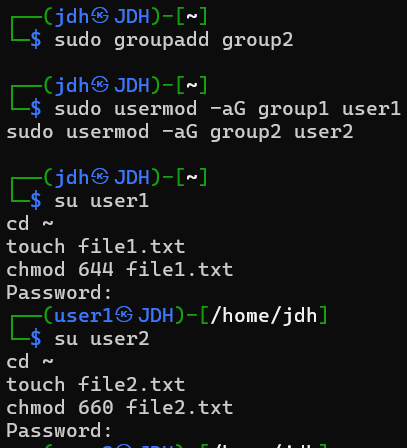
учетных записей и исследование влияния прав на файловые операции.

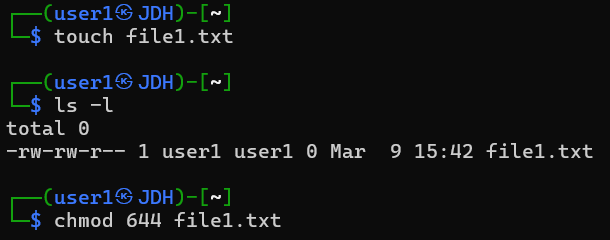
Изучение специфик фонового (background) и диалогового (foreground) режимов

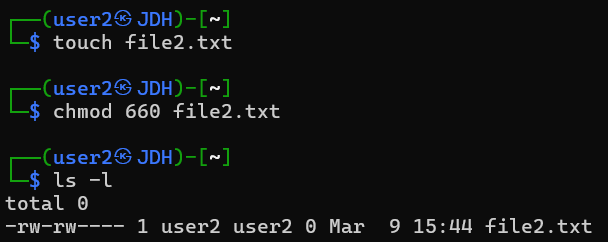
исполнения процессов и способов переключений между этими режимами.

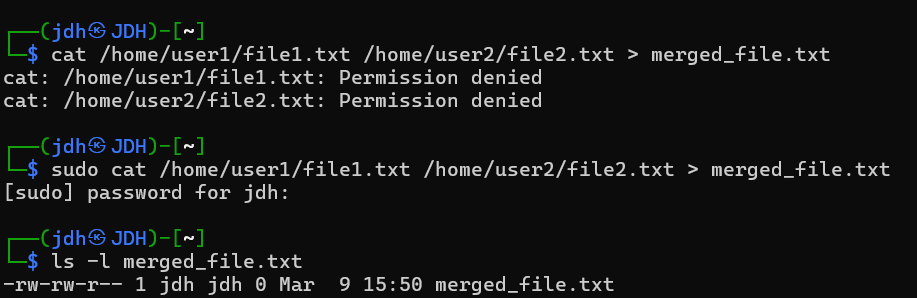
## Выполнение работы:

1.





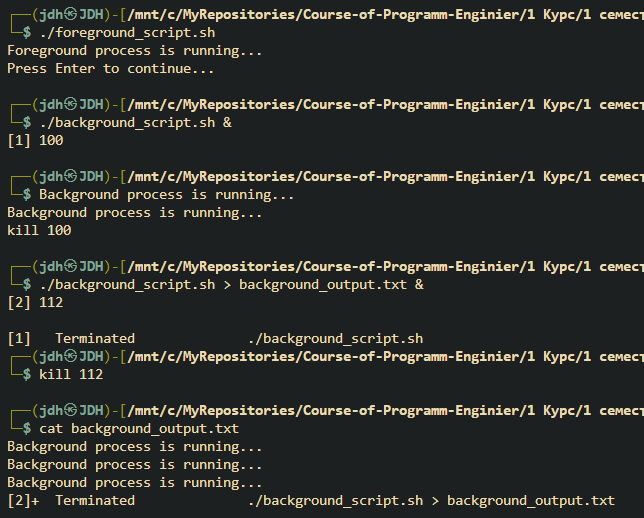




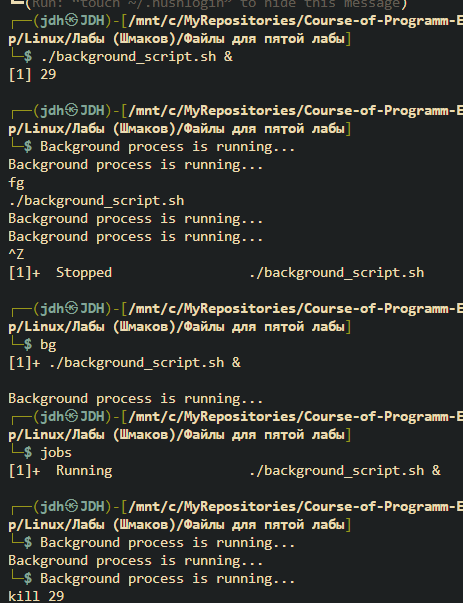
Права доступа на новый файл определяются **umask** (маской прав по умолчанию) текущей сессии или пользователя, который создаёт файл. Обычно umask по умолчанию для root — это 022, что означает, что новый файл будет создан с правами 644 (владелец — чтение и запись, группа и другие — только чтение).

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

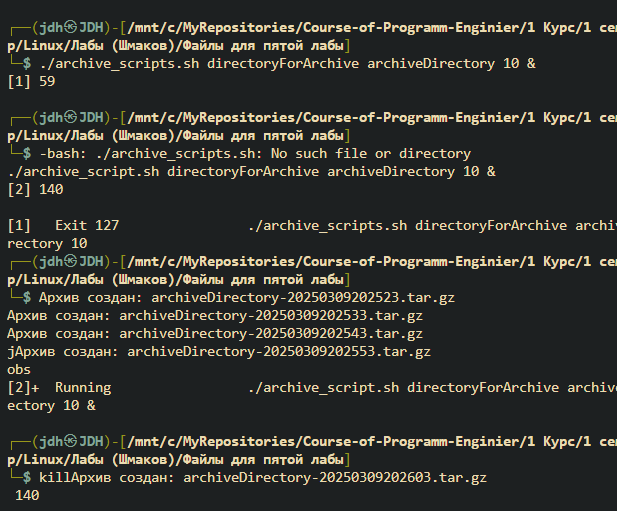
- в фоновом режиме процесс выводит результат в терминал (для наглядности)

- fg переводит в диалоговый режимам

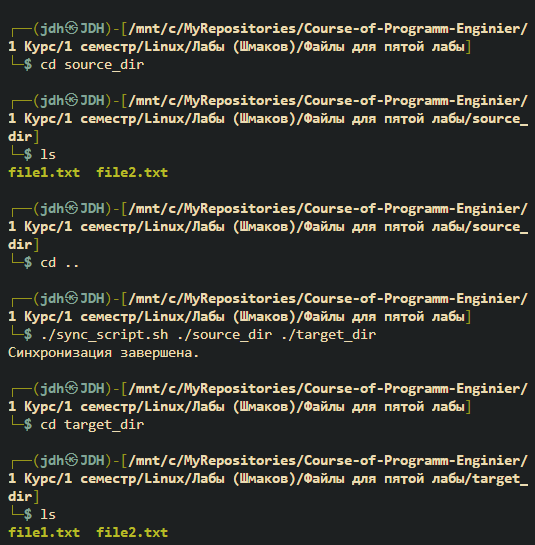
- bg переводит в фоновый режим (как запуск с &)

- 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 №7

## Цель работы

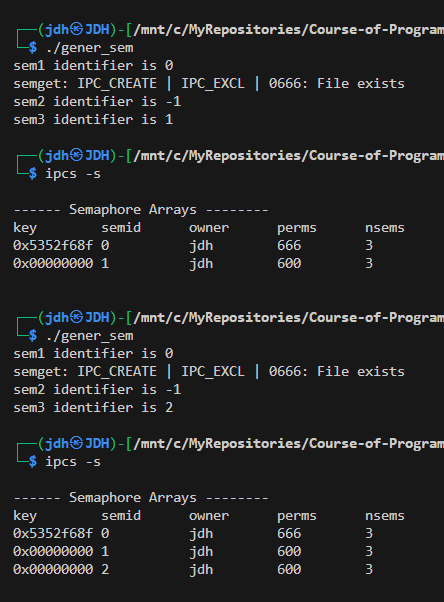
Освоение семафоров (semaphores) как эффективного

средства синхронизации доступа процессов к разделяемым ресурсам

операционной системы.

## Выполнение работы

1.



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.

# Лабораторная работа №9

## Цель работы

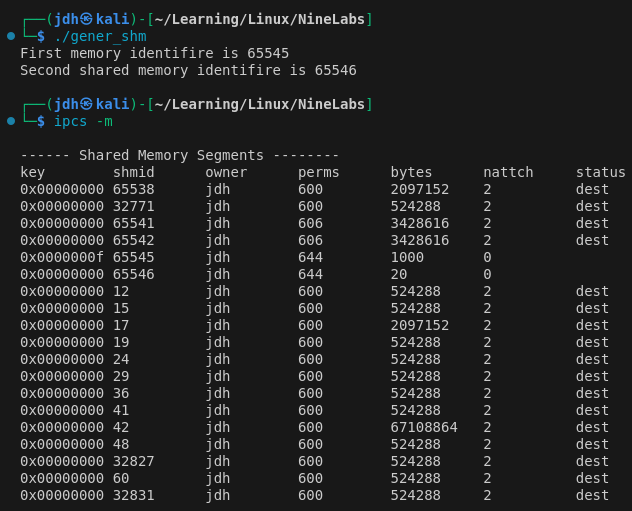
Использование для обмена данными разделяемой памяти

(shared memory) – самого быстрого средства межпроцессного взаимодействия в

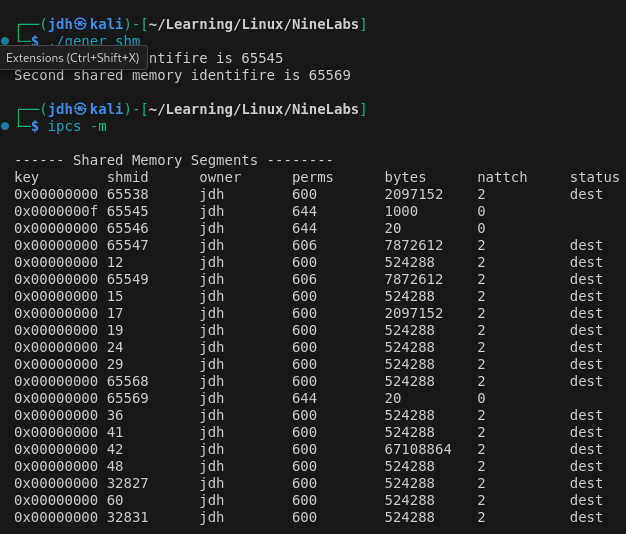
Linux.

## Последовательность выполнения работы

1.   
Первый запуск:

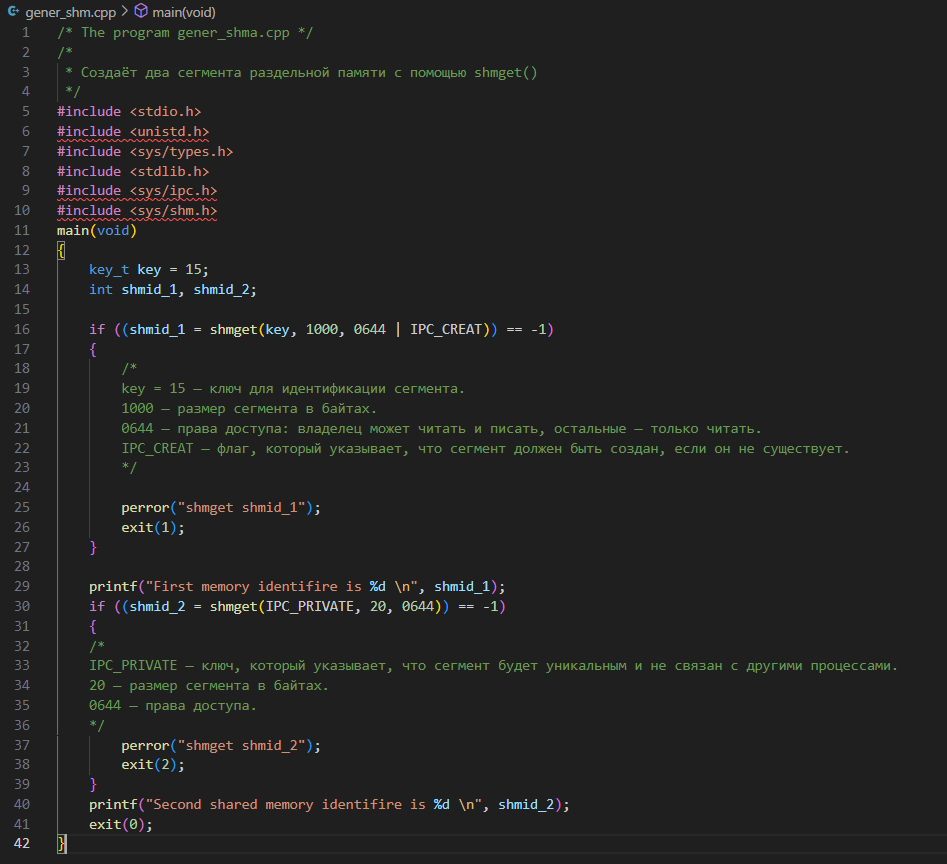


Второй запуск:

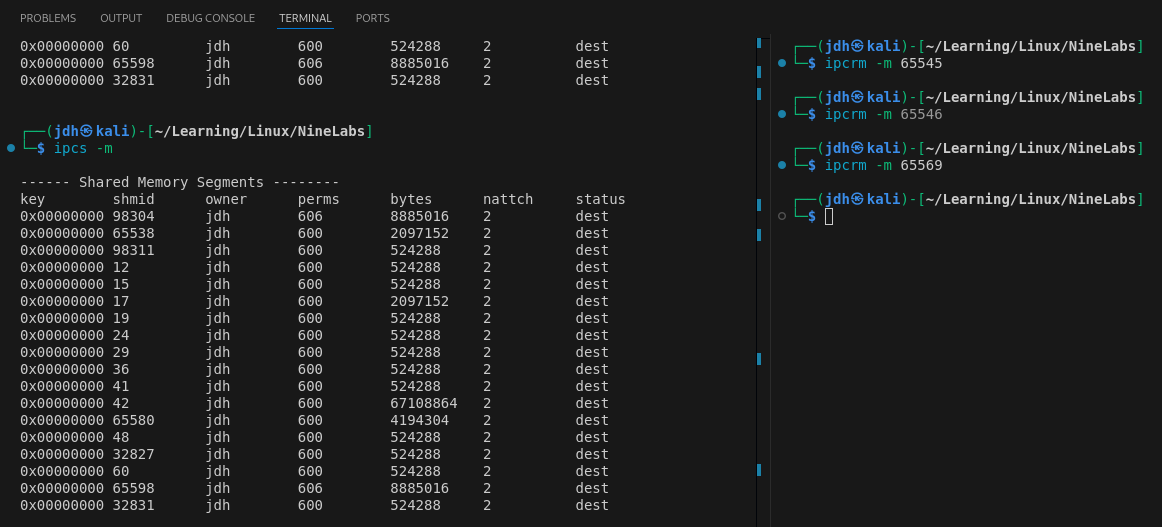


В программе создаётся два сегмента памяти. Первый — использует флаг IPC\_CREAT – такой флаг позволяет создать новый сегмент памяти по ключу, если его нет ИЛИ получить идентификатор сегмента, если он уже был создан.

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

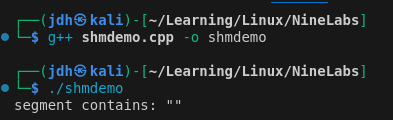


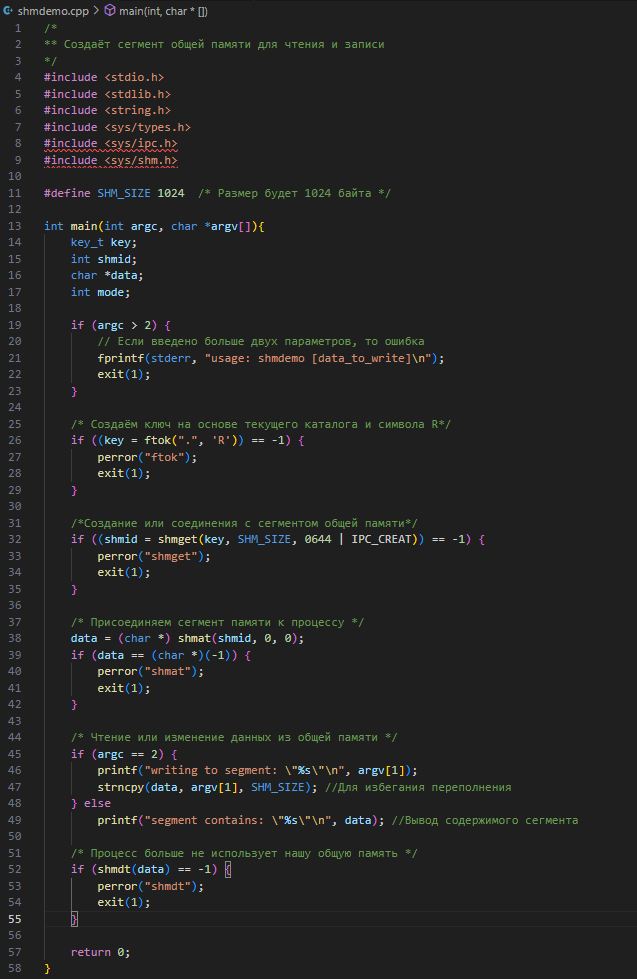
2. (Сначала правый терминал, потом левый)



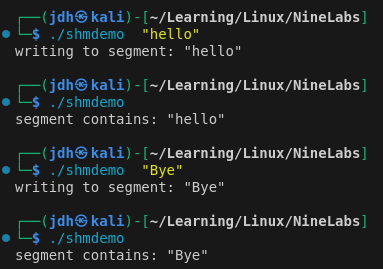
Удаление по ключу происходит аналогично

3.



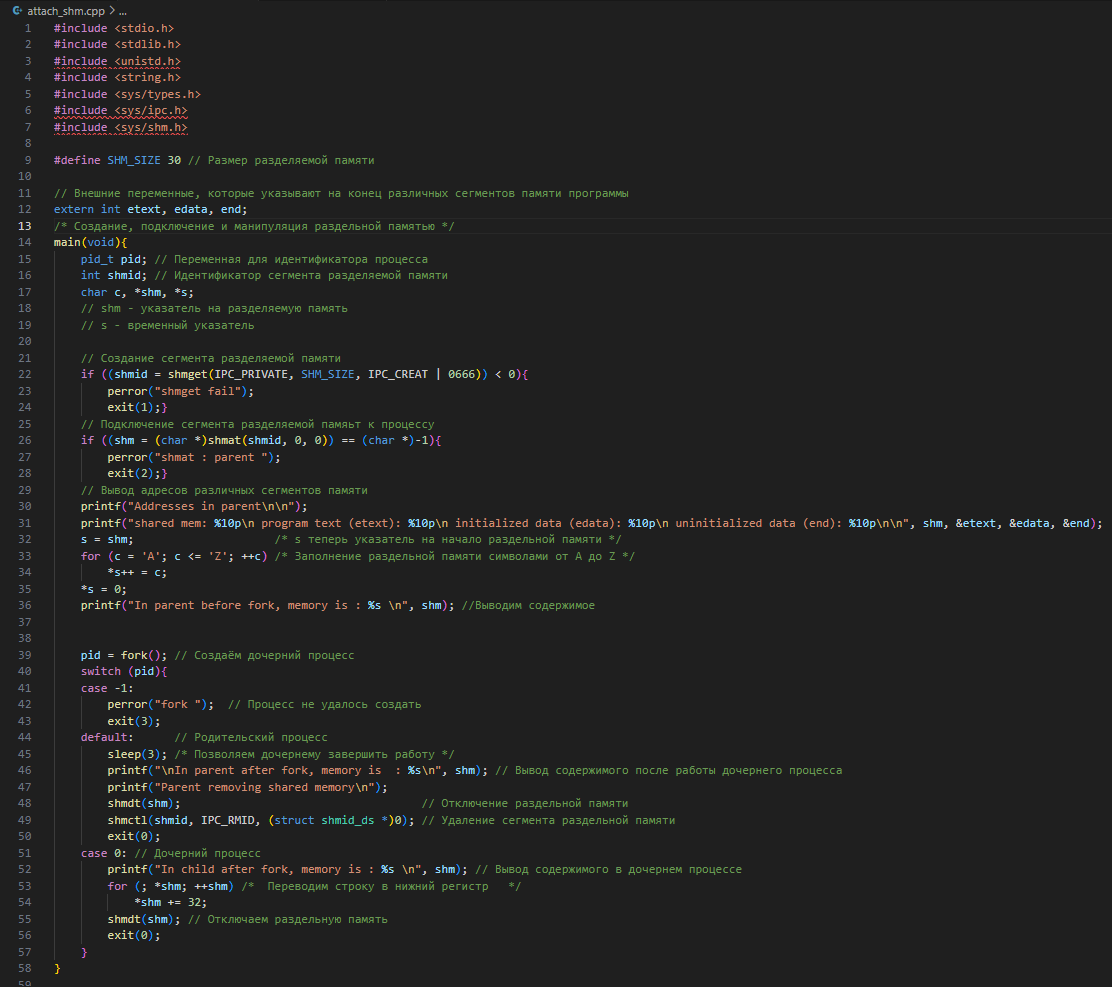


4.

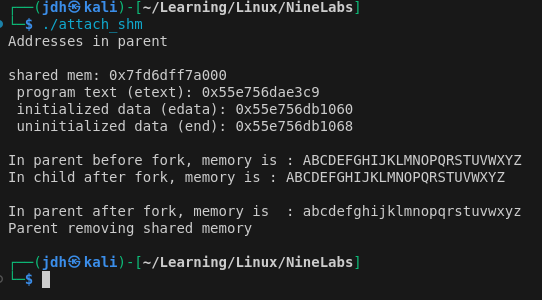


Удаление через ipcrm, как во втором шаге

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

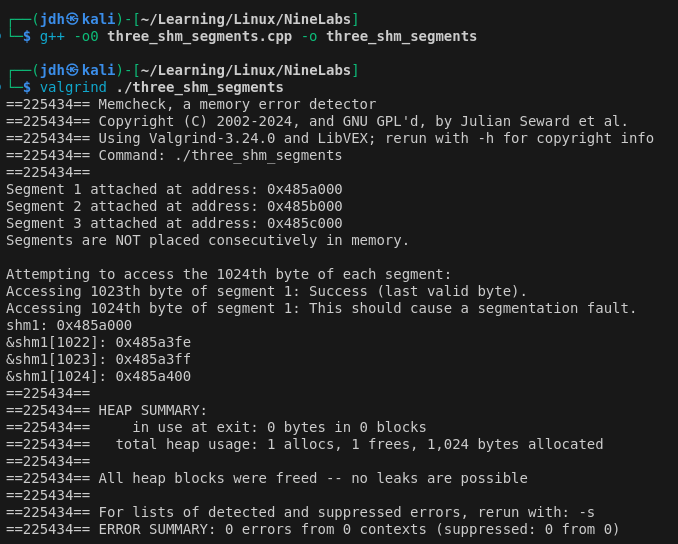
    shmctl(shmid1, IPC\_RMID, NULL);

    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.