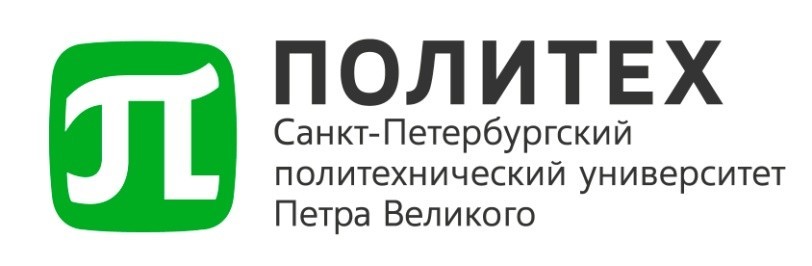
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

***«*САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»** ВШ программной инженерии



## Отчёт по выполнению лабораторных работ по дисциплине «Системное программное обеспечение GNU/Linux»»

Студенты гр. № 3530202/80202 Шемета К.Н.

Козлова Е.А.

Руководитель Шмаков В. Э.

Санкт-Петербург 2021 г.

## Оглавление

Оглавление

[Отчёт по выполнению лабораторных работ по дисциплине «Системное программное обеспечение GNU/Linux»» 1](#_Toc85818200)

[Оглавление 2](#_Toc85818201)

[Лабораторная работа № 1. Базовые команды 3](#_Toc85818202)

[Рисунок 1: Использование команд who, pwd, ls, cd, mkdir 3](#_Toc85818203)

[Рисунок 2: Создание файлов 4](#_Toc85818204)

[Рисунок 3: Midnight Commander 4](#_Toc85818205)

[Рисунок 4: Использование команды cat 5](#_Toc85818206)

[Рисунок 5: Использование команды cp 6](#_Toc85818207)

[Рисунок 6: Использование команды find 6](#_Toc85818208)

[Рисунок 7: Использование команды link 6](#_Toc85818209)

[Рисунок 8: Darkroom 7](#_Toc85818210)

[Лабораторная работа № 2. Запуск и завершение процессов 8](#_Toc85818211)

[Лабораторная работа № 3. Программные каналы 13](#_Toc85818212)

[Лабораторная работа № 4. Командные файлы. Переменные окружения. 19](#_Toc85818213)

[Лабораторная работа № 5. Учетные записи. Фоновый и диалоговый режимы исполнения процессов 21](#_Toc85818214)

[Листинг 3: Background процесс 22](#_Toc85818215)

[Лабораторная работа № 6. Генерация и обработка сигналов 22](#_Toc85818216)

[Лабораторная работа № 7. Семафоры и синхронизация 30](#_Toc85818217)

[Рисунок 1: Демонстрация semrm во время выполнения semdemo 31](#_Toc85818218)

[Листинг 1: Фрагмент изменённого текста программы semdemo.cpp 32](#_Toc85818219)

[Листинг 2: Код программы monitor\_sem.cpp 33](#_Toc85818220)

[Лабораторная работа № 8. Обмен через очереди сообщений 33](#_Toc85818221)

[Лабораторная работа № 9. Работа с разделяемой памятью 36](#_Toc85818222)

[Лабораторная работа № 10. Создание соединений на сокетах 41](#_Toc85818223)

[Лабораторная работа № 11. Взаимодействие процессов по сети 43](#_Toc85818224)

[Рисунок 1: Резльтат выполнения команды netstat -m a | grep 1066 44](#_Toc85818225)

[Рисунок 2: Запуск сервера и соединение клиента 44](#_Toc85818226)

[Рисунок 3: Завершение работы сервера 45](#_Toc85818227)

# Лабораторная работа № 1. Базовые команды

**Цель работы.** Освоение минимального набора базовых команд операционной системы

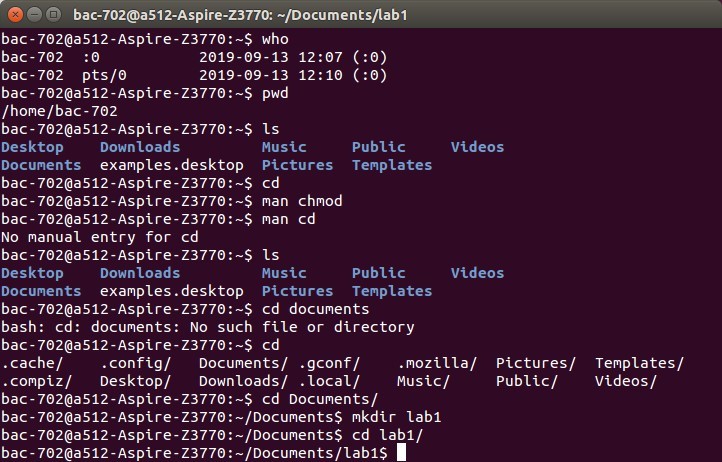
Linux, знакомство с файловой системой, особенностями прав доступа, получение первичных навыков работы под Linux.

**Ход работы.**

***1. Необходимо открыть терминал (Ctrl + Alt + t), выполнить команды shell (pwd , Ctrl + Alt + t), выполнить команды shell (Ctrl + Alt + t), выполнить команды shell (pwd , pwd , who , ls , cd , mkdir , rm , chmod) и проанализировать результат их выполнения.***

В терминале были выполнены следующие команды:

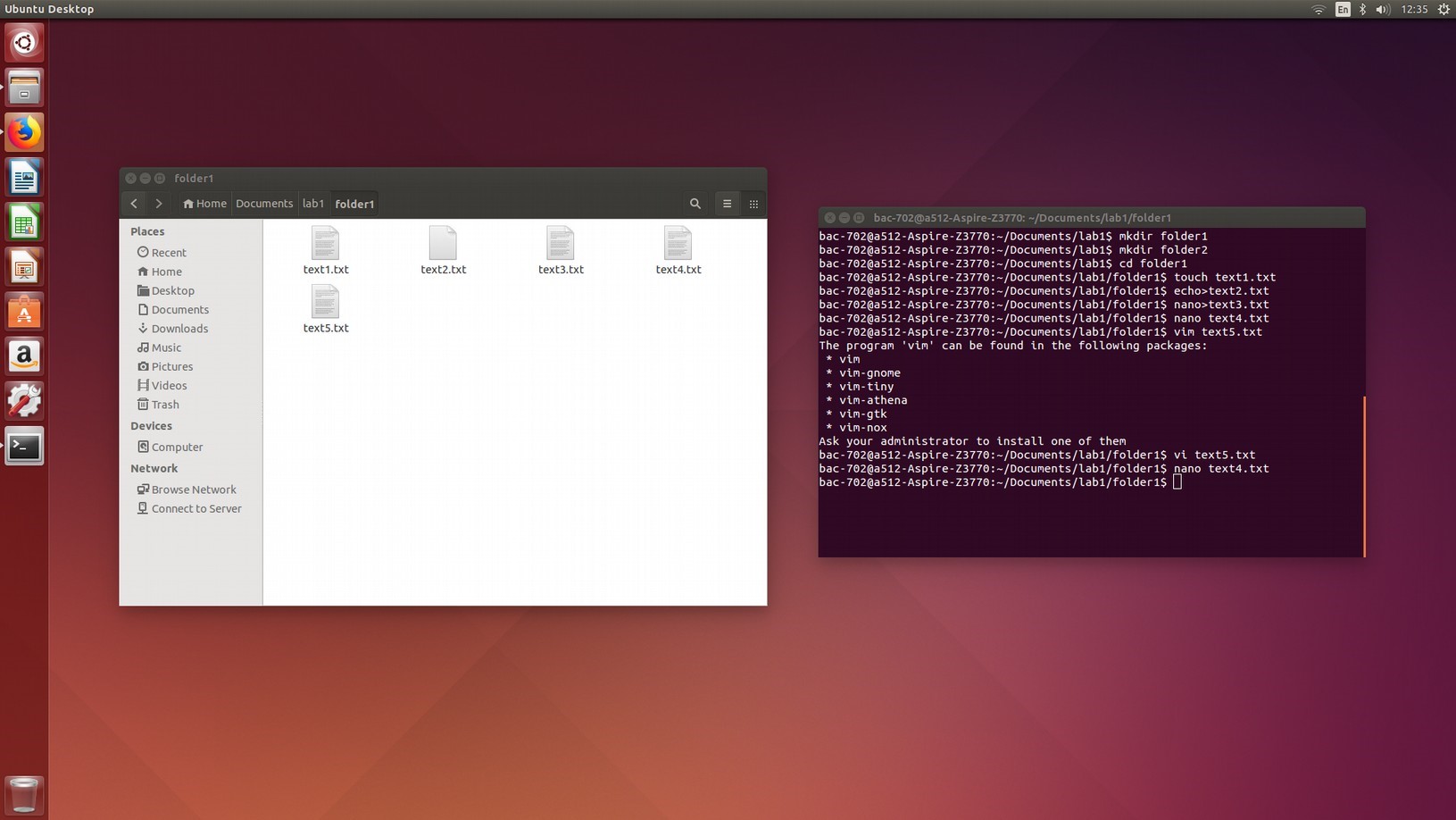
* who – показывает информацию о пользователях, подключенных к системе.
* pwd – печатает полный путь текущего каталога.
* ls – печатает содержимое каталога.
* cd – измененяет текущего каталога.
* mkdir – создаёт новый каталог.
* rm – удаляет файлы или каталоги.
* chmod – изменяет права доступа к файлам или каталогам.



### Рисунок 1: Использование команд who, pwd, ls, cd, mkdir

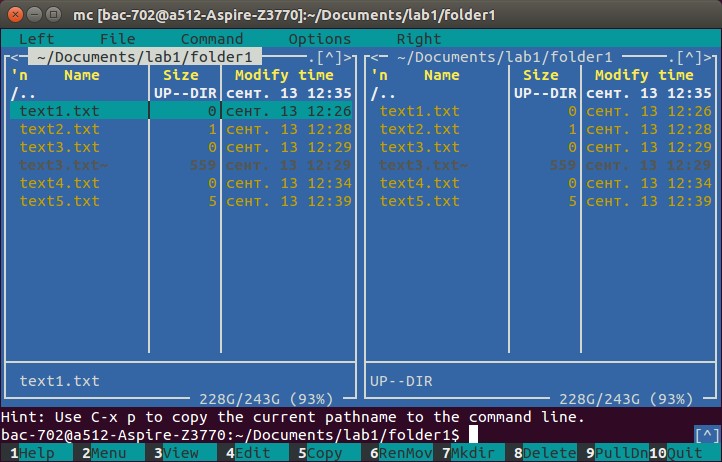
***2. Необходимо создать дерево каталогов глубиной вложения до трех уровней, а в самих каталогах создать текстовые файлы; применить различные способы создания новых файлов.***

Были созданы два каталога folder1 и folder2 с помощью команды mkdir. Текстовые файлы были созданы с помощью команды touch и запуска текстовых редакторов (nano, vi).nano, vi).



### Рисунок 2: Создание файлов

***3. Необходимо запустить с терминала Midnight Commander вводом команды mc и ознакомиться с его основными возможностями по работе с файловой системой.***

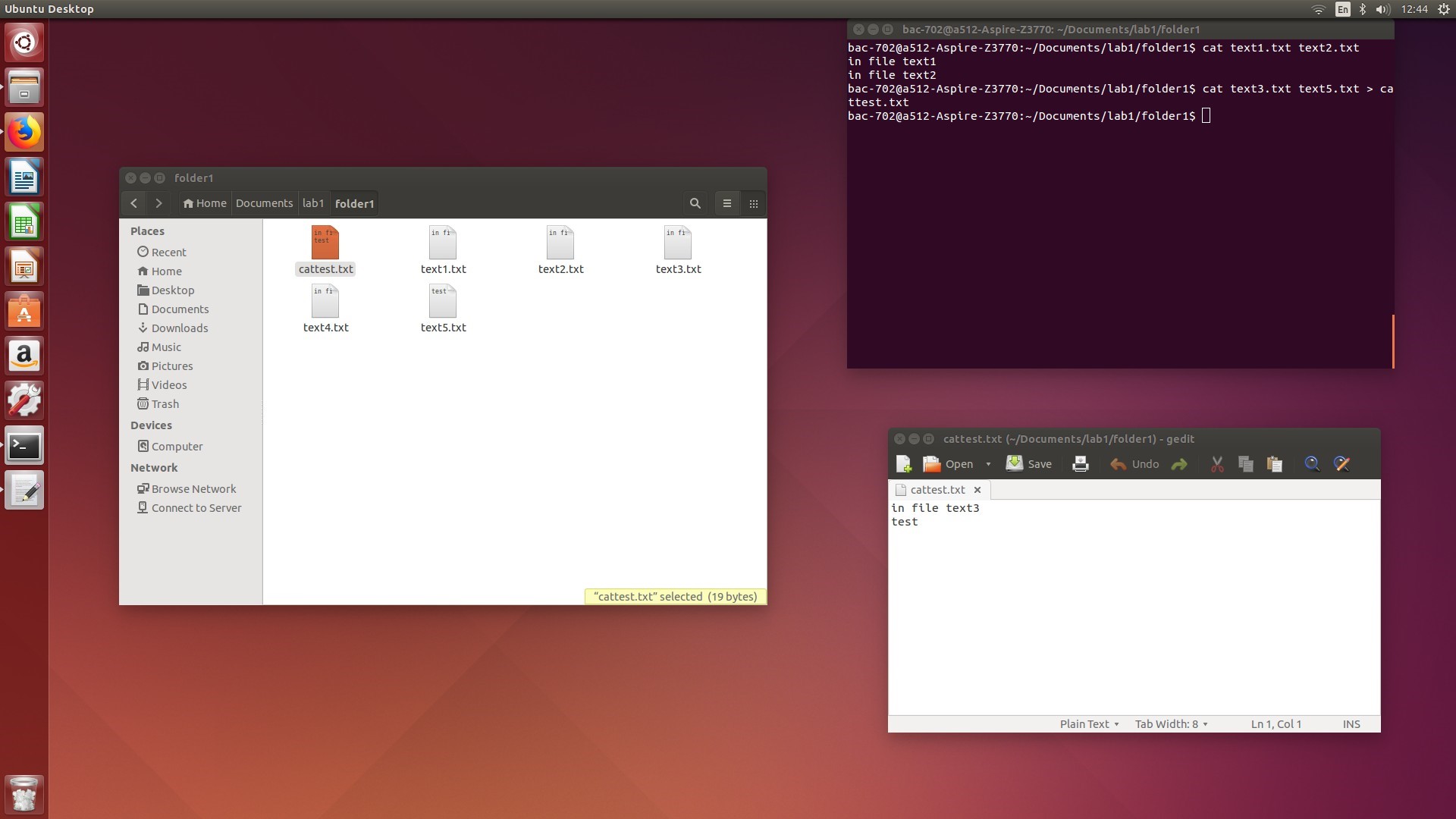


### Рисунок 3: Midnight Commander

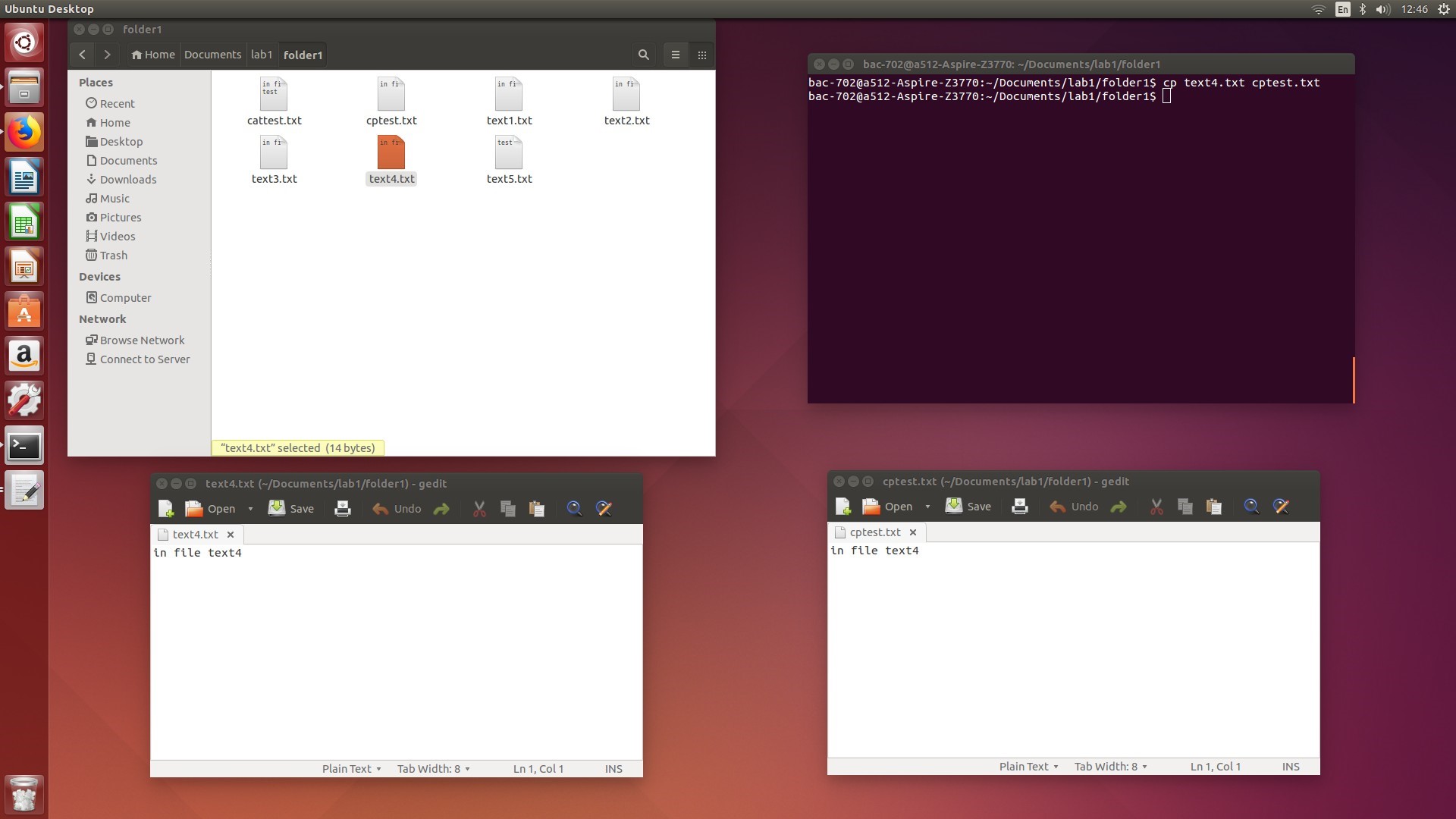
***4. Необходимо выполнить на терминале вторую серию команд cat, cp, find, link, chmod, применяя их вместе с текстовыми файлами, созданными на предыдущем шаге.***

В терминале были выполнены следующие команды:

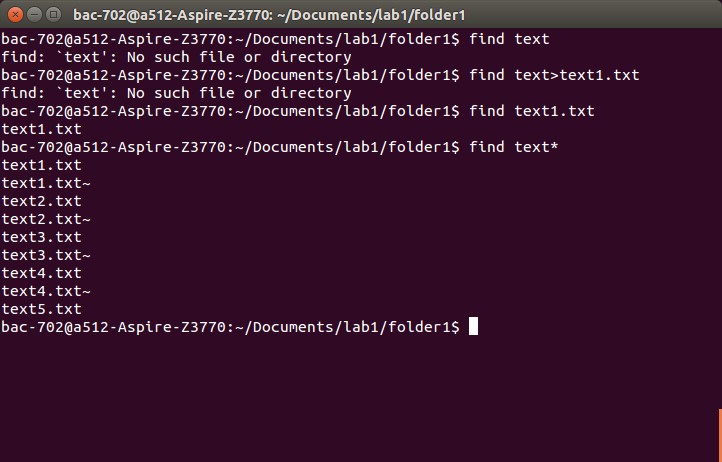
* cat – объединение содержимого двух файлов и печать результата. На рис
* cp – копирование файлов и каталогов.
* find – поиск файлов в каталоге.
* link – создание ссылки на файл.
* chmod – изменение прав доступа к файлам и каталогам



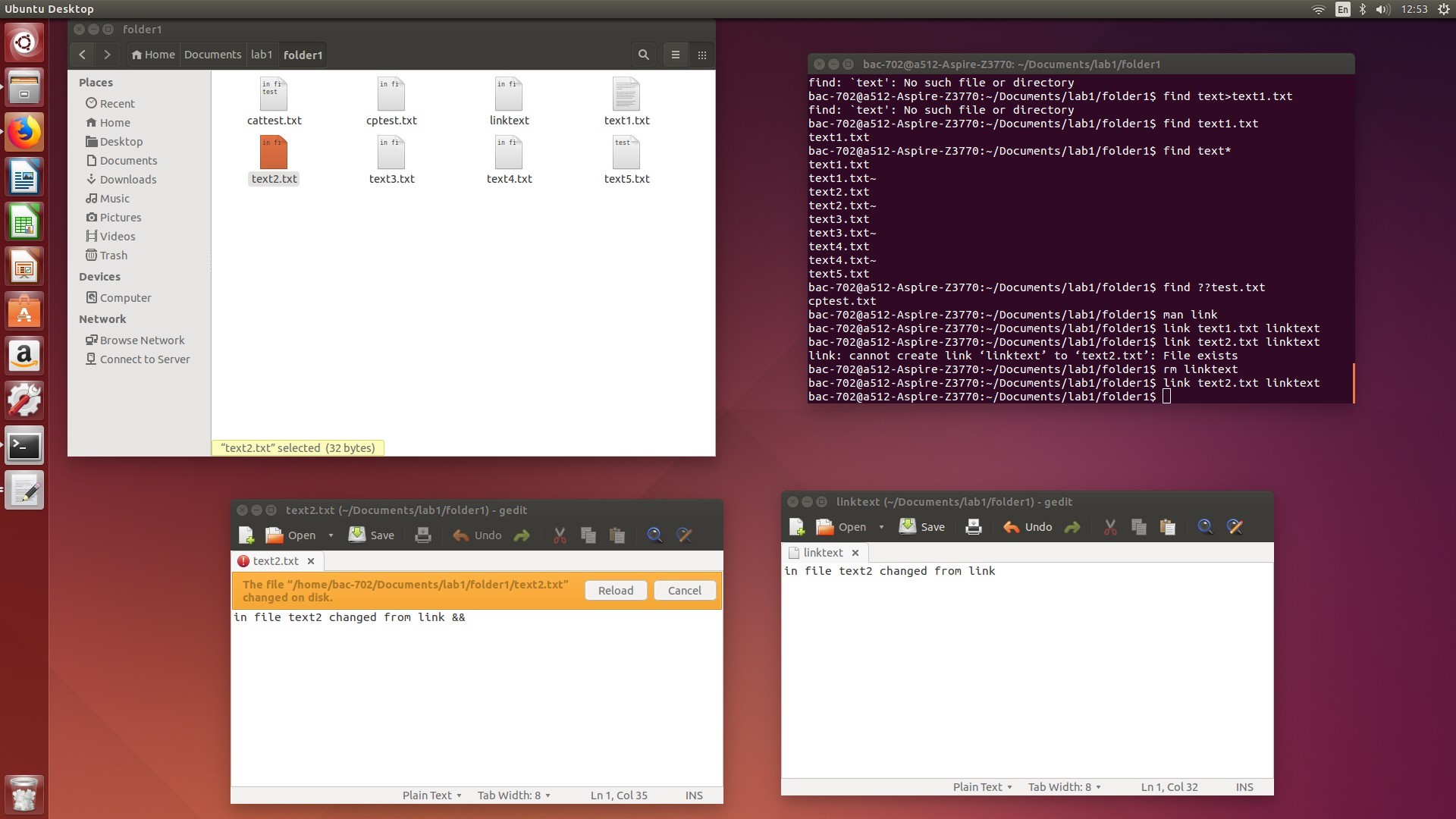
### Рисунок 4: Использование команды cat



### Рисунок 5: Использование команды cp



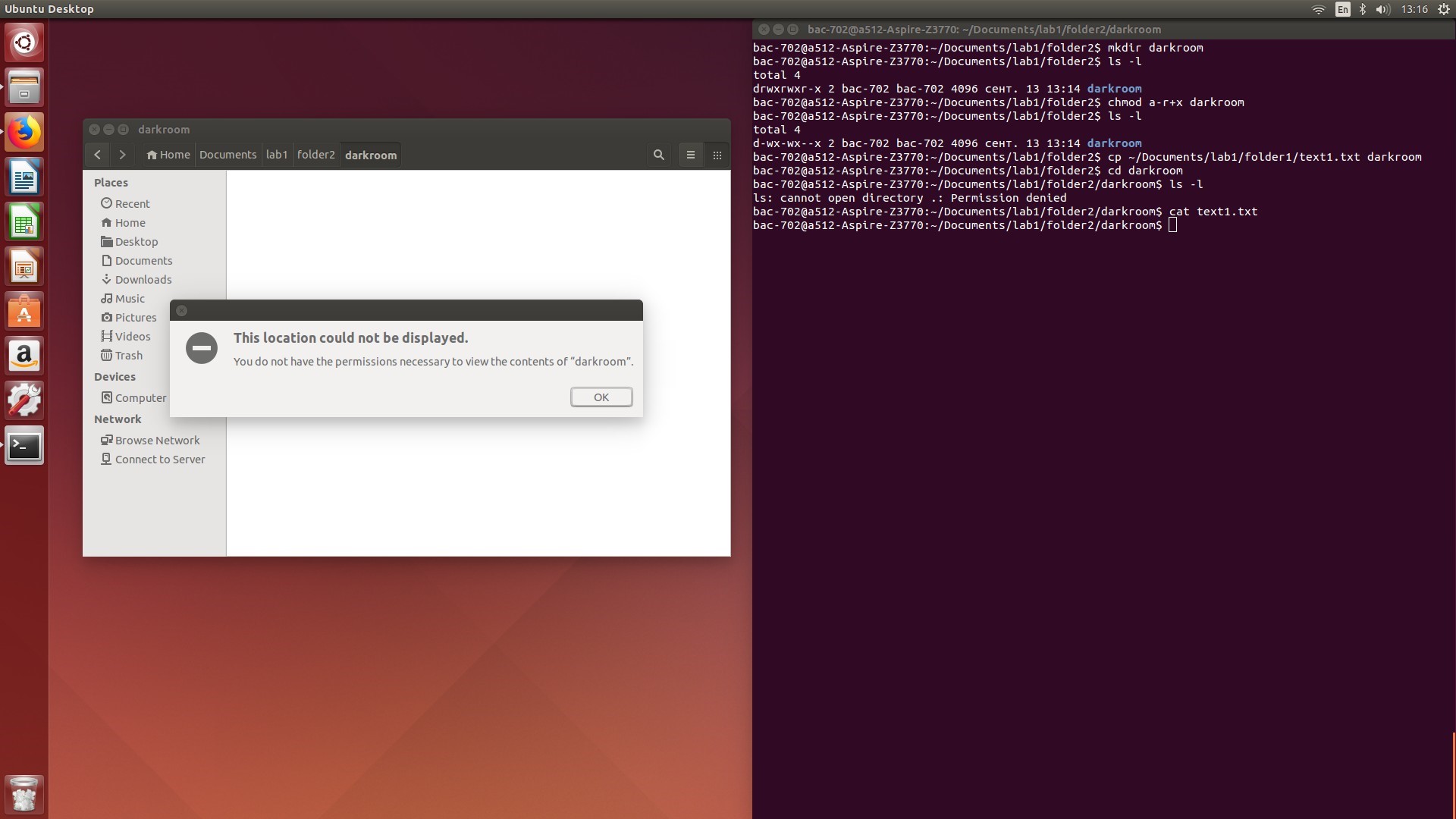
### Рисунок 6: Использование команды find



### Рисунок 7: Использование команды link

***5. Попытайтесь создать на своем дереве какой-нибудь каталог с правами доступа, аналогичными каталогу darkroom , рассмотренному в лекциях.***

“Тёмный" каталог – каталог, файлы которого доступны только в случае, если пользователь заранее знает их имена, поскольку получение списка файлов таких каталогов запрещено. На рис. 8 представлен процесс создания такого каталога, в который скопирован файл text1.txt. Видно, что получение списка файлов запрещено (nano, vi).результат команды ls), однако зная полное имя файла, можно получить к нему доступ. Например, можно применить команду cat.



### Рисунок 8: Darkroom

**Выводы.**

В ходе работы были освоены такие команды ОС Linux, как *pwd , who , ls , cd , mkdir , rm , chmod, cat, cp, find, link,* была рассмотрена файловая система и права доступа к файлам и каталогам.

# Лабораторная работа № 2. Запуск и завершение процессов

**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.

**Ход работы.**

1. ***Модифицируйте программу forkdemo.cpp, так чтобы ввод/вывод на терминал отсутствовал, а при проходе по циклу была временная задержка, например sleep***

***(Ctrl + Alt + t), выполнить команды shell (pwd , 7). Запустите эту программу в фоновом режиме (Ctrl + Alt + t), выполнить команды shell (pwd , background), введя при запуске***

***символ & после пробела и зафиксировав значение PID, назначенное системой фоновому процессу при запуске. Выполните на терминале команды ps , top , uptime , pstre . Снимите свой фоновый процесс командой kill с соответствующими параметрами.***

Код программы:

#include<stdio.h>

#include<sys/types.h>

#include<unistd.h>

int main(nano, vi).) { if(nano, vi).fork(nano, vi).)) {

for(nano, vi).size\_t i = 0; i < 1000; ++i) {

sleep(nano, vi).7);

}

} else {

for(nano, vi).size\_t i = 0; i < 1000; ++i) {

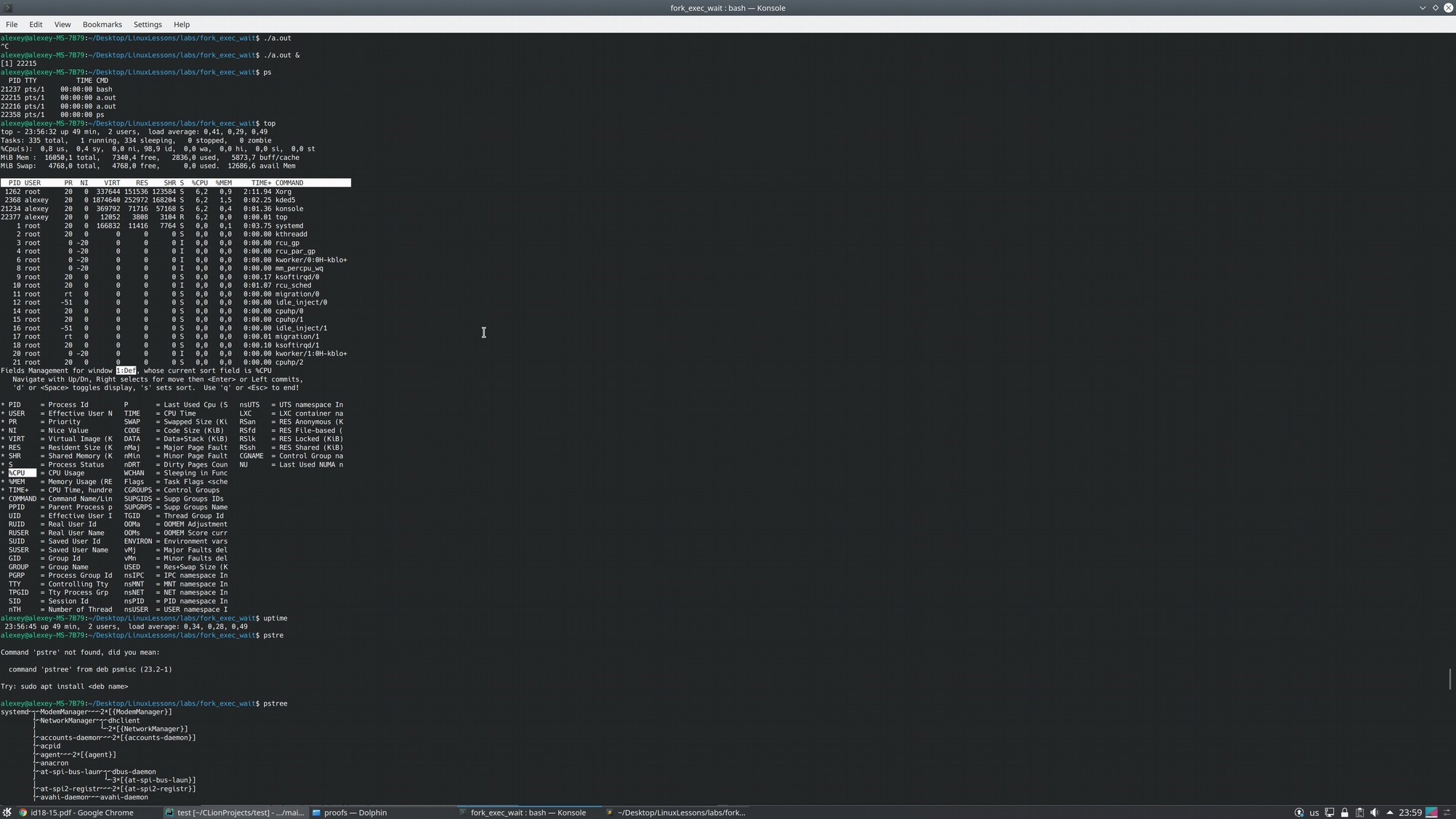
sleep(nano, vi).7);

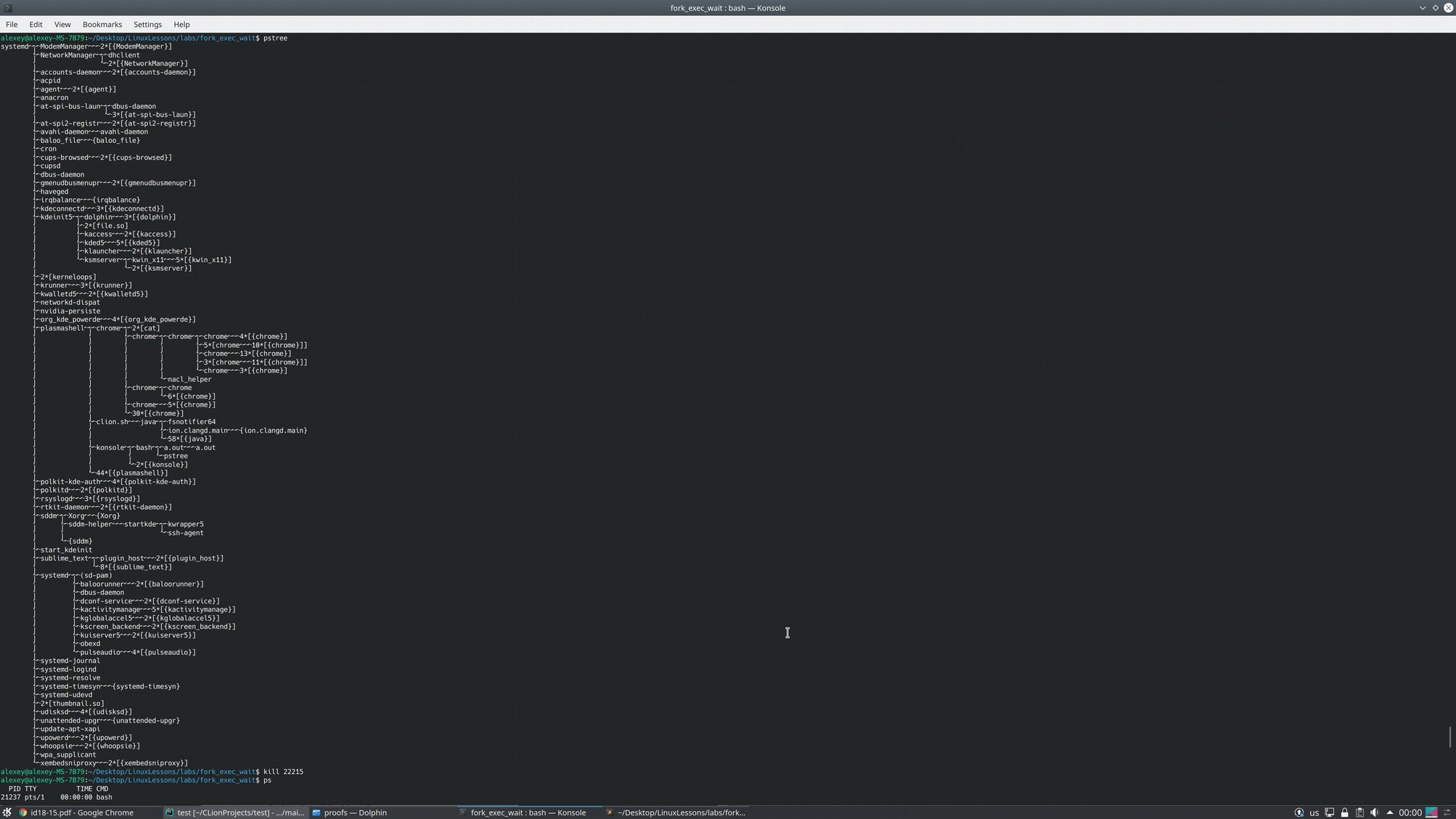
}

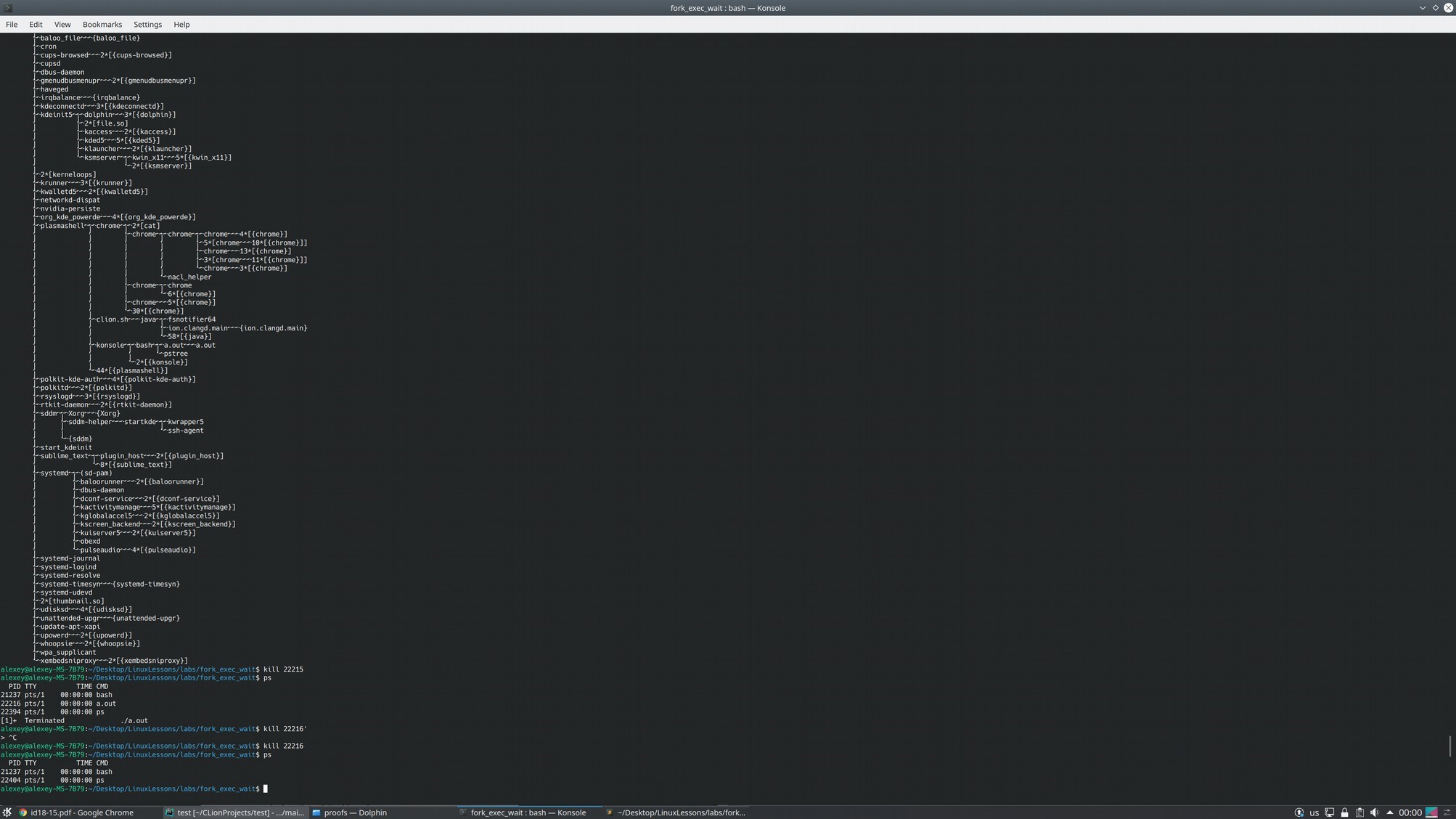
}

return 0;

} а







***2. Исследуйте, что произойдет, если процесс-потомок сменит текущий каталог, будет ли изменен текущий каталог для родителя? Создайте программу, подтверждающую ответ***

Нет, не будет. Для подтверждения этого, был создан файл txt, в котором записана строка “ChildFatheraaaa”, затем создан каталог NewCat, в котором лежит файл txt со строкой “ sssssssssssssss”. Потомок меняет директорию, читает файл txt и выводит sssss, затем просыпается родитель, читает файл txt и выводит Child.

Код программы:

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#include <fcntl.h> #include <errno.h> char buf[256];

int main(nano, vi).) {

const pid\_t pid = fork(nano, vi).); if(nano, vi).!pid) {

chdir(nano, vi)."newCat");

const int desc = open(nano, vi)."txt", 0);

read(nano, vi).desc, buf, 5); write(nano, vi).1, buf, 5); sleep(nano, vi).5);

\_exit(nano, vi).0);

} else if(nano, vi).pid > 0) {

sleep(nano, vi).4);

const int desc = open(nano, vi)."txt", 0);

read(nano, vi).desc, buf, 5);

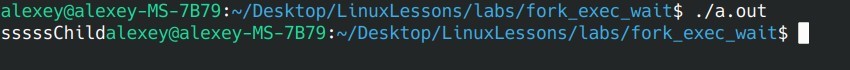
write(nano, vi).1, buf, 5);

\_exit(nano, vi).0);

}

printf(nano, vi)."Got an error %d", errno);

return 0; }



***3. 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.***

Был создан файл txt, в котором записана строка “ ChildFatheraaaa”. Дальше потомок читает 5 символов, выводит их на консоль, затем просыпается родитель, читает 6 символов и выводит их на консоль. В итоге окзалось, что у них одно смещение текстового файла.

Код программы:

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#include <fcntl.h> #include <errno.h> char buf[256];

int main(nano, vi).) {

const int desc = open(nano, vi)."txt", 0); if(nano, vi).~desc) {

const pid\_t pid = fork(nano, vi).); if(nano, vi).!pid) { read(nano, vi).desc, buf, 5);

write(nano, vi).1, buf, 5);

sleep(nano, vi).8);

\_exit(nano, vi).0);

} else if(nano, vi).pid > 0) {

sleep(nano, vi).4);

read(nano, vi).desc, buf, 6);

write(nano, vi).1, buf, 6);

\_exit(nano, vi).0);

}

}

printf(nano, vi)."Got an error %d", errno);

return 0; }



**Выводы.**

В ходе работы была освоена характерная для Linux схема порождения и завершения процессов, изучены вызвовы fork и семейство вызовов exec. Было обнаружено, что родитель и потомок разделяют дескриптор. Также было обнаружено, что если потомок меняет каталог, то это не влияет на родительский процесс.

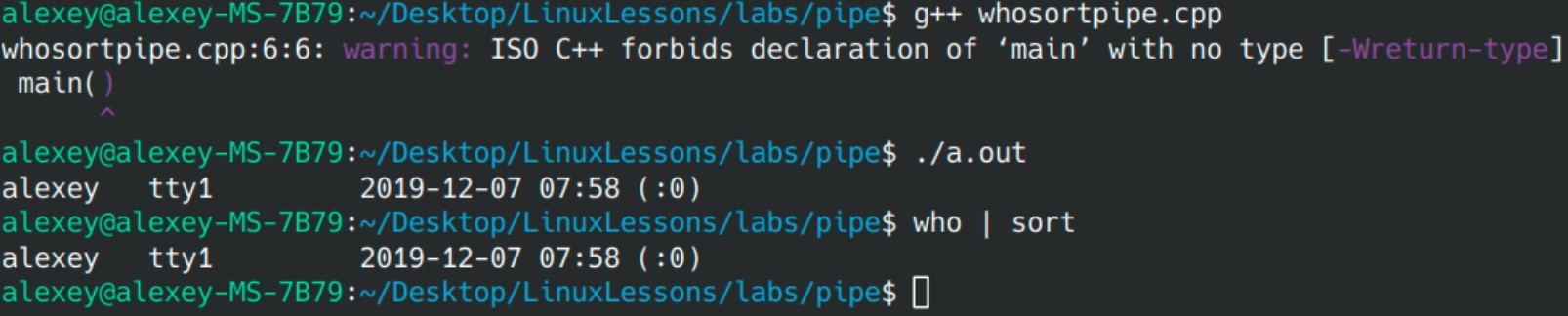
# Лабораторная работа № 3. Программные каналы

**Цель работы.** Изучение конвейеров (nano, vi).pipes, программных каналов), как простейшего средства коммуникации запущенных процессов. Исследование различных способов организации каналов и их сопоставление

**Ход работы.**

1. ***Скомпилируйте и выполните программу whosortpipe.cpp . Сопоставьте результат выполнения программы с выполнением этих же двух команд из shell в конвейерном режиме (Ctrl + Alt + t), выполнить команды shell (pwd , | ).***

Результаты идентичны.

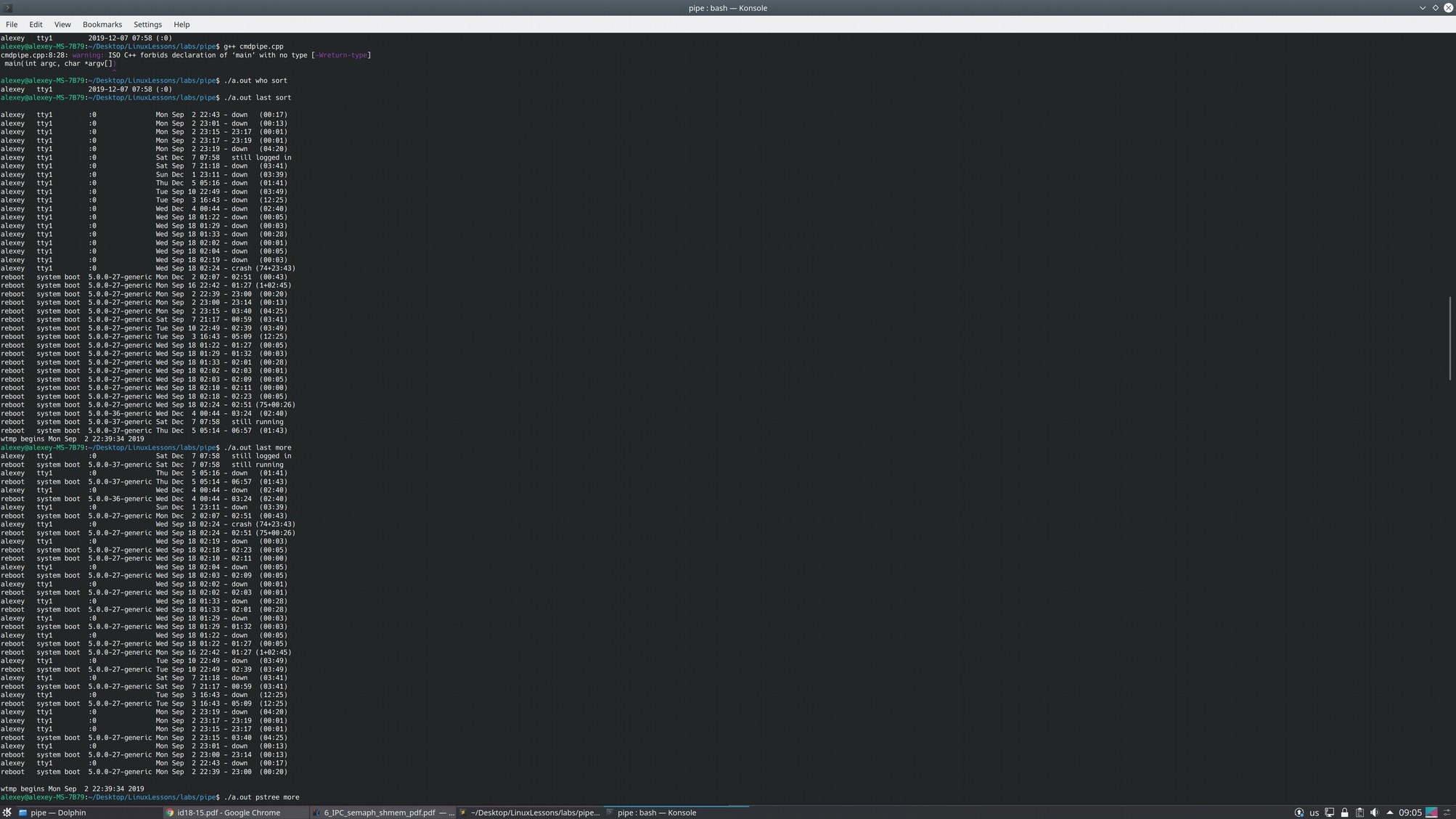


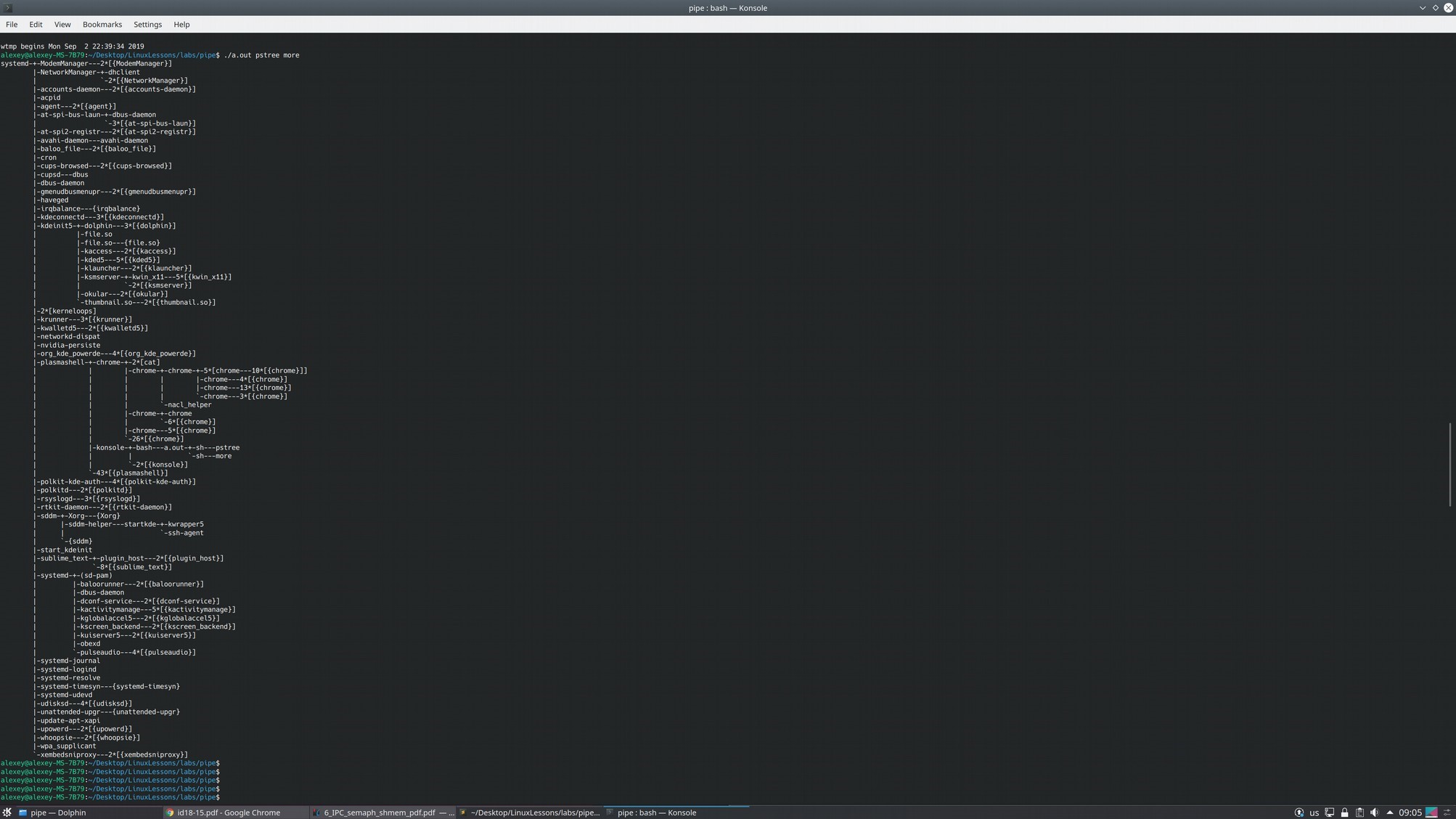
1. ***Программу cmdpipe.cpp запускайте после компиляции, задавая ей при стартах в качестве параметров командной строки пары команд shell для конвейеризации***

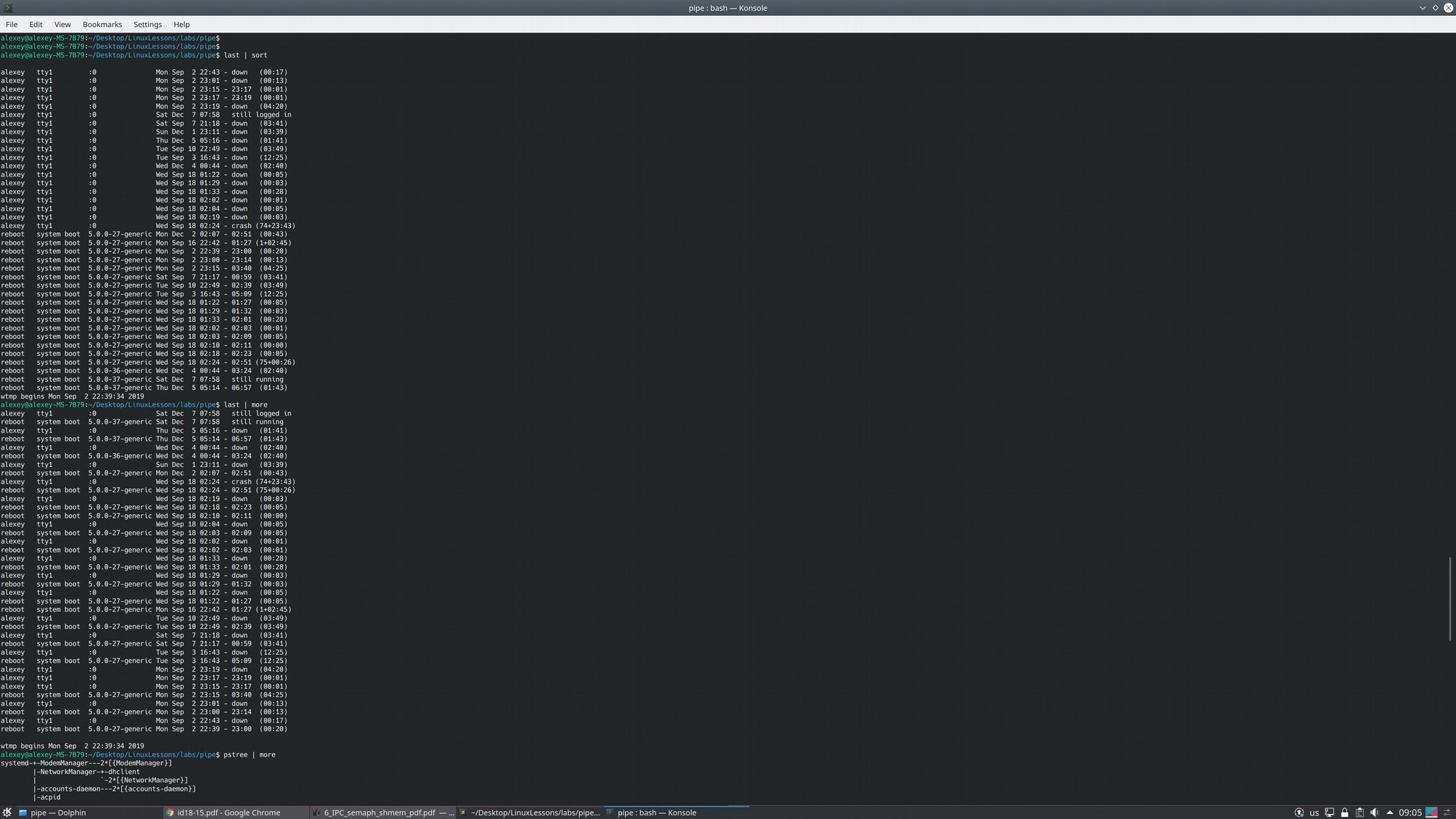
***(Ctrl + Alt + t), выполнить команды shell (pwd , who и sort ; last и sort ; last и more ; pstree и more). Сопоставьте результаты***

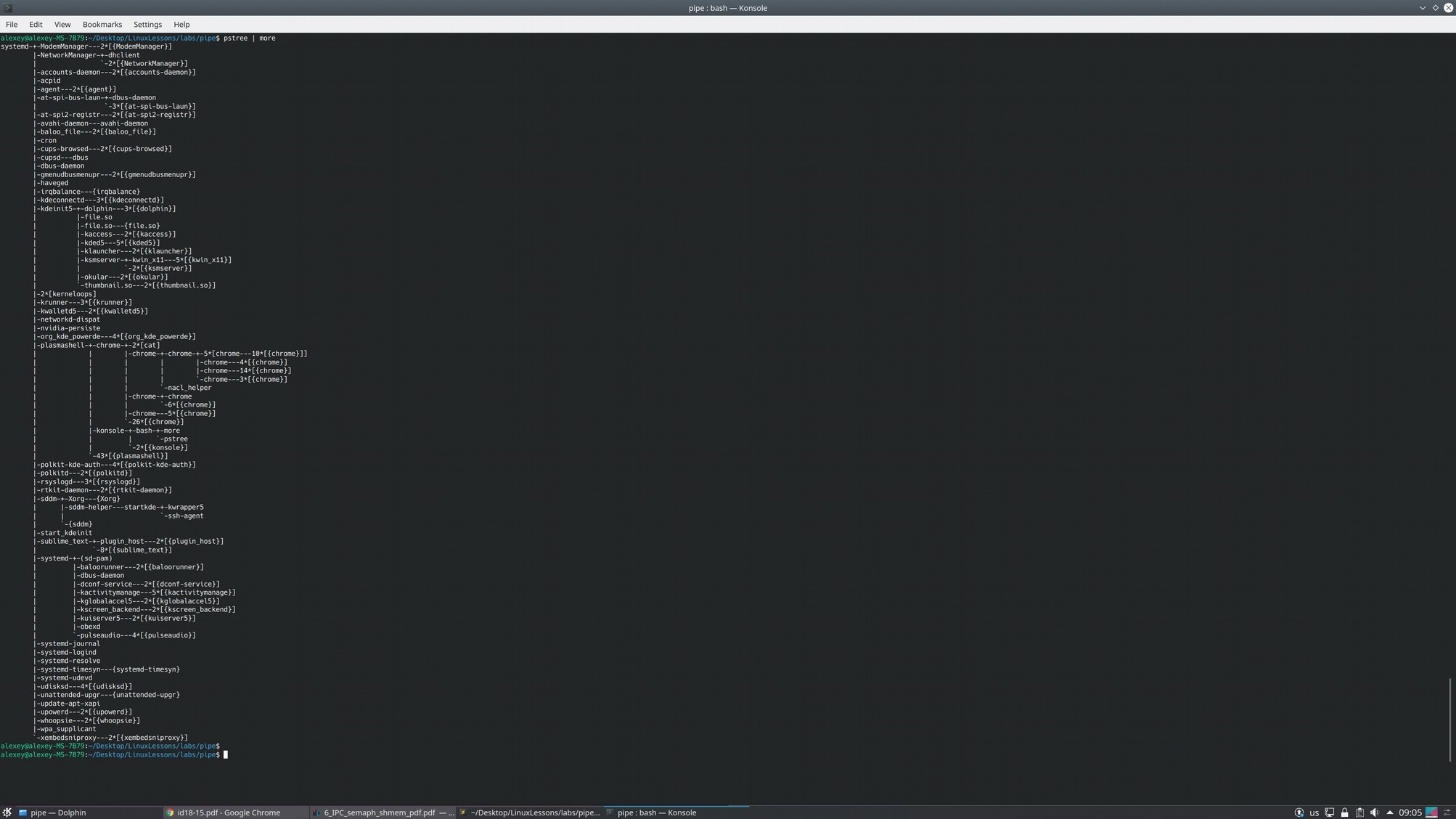
***запусков программы с выполнением тех же пар команд из shell в конвейерном режиме.***

Результаты идентичны.









1. ***Можно ли с помощью вызова popen(Ctrl + Alt + t), выполнить команды shell (pwd , ) создать программу, организующую конвейер из трех команд shell , передаваемых ей в качестве параметров командной строки при запуске? Если да, то создайте такую программу, если нет, дайте обоснованный ответ, почему нельзя.***

Нет, нельзя, так как канал задается однонаправленным и аргумент type может указать только на режим чтения или записи, но не на оба одновременно.

1. ***Напишите программу (Ctrl + Alt + t), выполнить команды shell (pwd , например, на основе вызовов pipe(Ctrl + Alt + t), выполнить команды shell (pwd , )), воспринимающую варьируемое количество команд, передаваемых ей при запуске в качестве параметров. Каждая последующая команда должна быть соединена с предыдущей с помощью конвейера. Так, при запуске программы $ ./a.out last sort more должны выполняться действия, эквивалентные запуску команд из shell : $ last | sort | more.***

Код программы:

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<sys/types.h>

#include<limits.h> #include<sys/wait.h>

int main(nano, vi).int argc, char \* argv[]) {

if(nano, vi).argc < 3) {

fprintf(nano, vi).stderr, "Usage %s cmd1 cmd2 cmd3 ... \n", argv[0]); return 1;

}

int \* fds = (nano, vi).int \*)malloc(nano, vi).sizeof(nano, vi).\*fds) \* (nano, vi).argc - 2) \* 2);

pid\_t pId;

pipe(nano, vi).fds); pId = fork(nano, vi).);

for(nano, vi).int i = 0; i < (nano, vi).argc - 2); ++i) { if(nano, vi).!pId) {

dup2(nano, vi).fds[i\*2+1], 1); close(nano, vi).fds[i\*2]);

execlp(nano, vi).argv[i+1], argv[i+1], NULL);

} else {

if(nano, vi).i < (nano, vi).argc - 3)) {

pipe(nano, vi).&fds[(nano, vi).i + 1) \* 2]);

}

pId = fork(nano, vi).);

if(nano, vi).!pId) { dup2(nano, vi).fds[i\*2], 0);

close(nano, vi).fds[i\*2+1]);

} else {

close(nano, vi).fds[i\*2]); close(nano, vi).fds[i\*2+1]);

}

}

}

if(nano, vi).!pId) {

execlp(nano, vi).argv[argc - 1], argv[argc - 1], NULL);

} else {

for(nano, vi).int i = 0; i < (nano, vi).argc - 1); ++i) {

wait(nano, vi).0);

}

}

free(nano, vi).fds);

return 0;

}



1. ***Разберите 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 результаты функционирования данной системы и ее недостатки.***

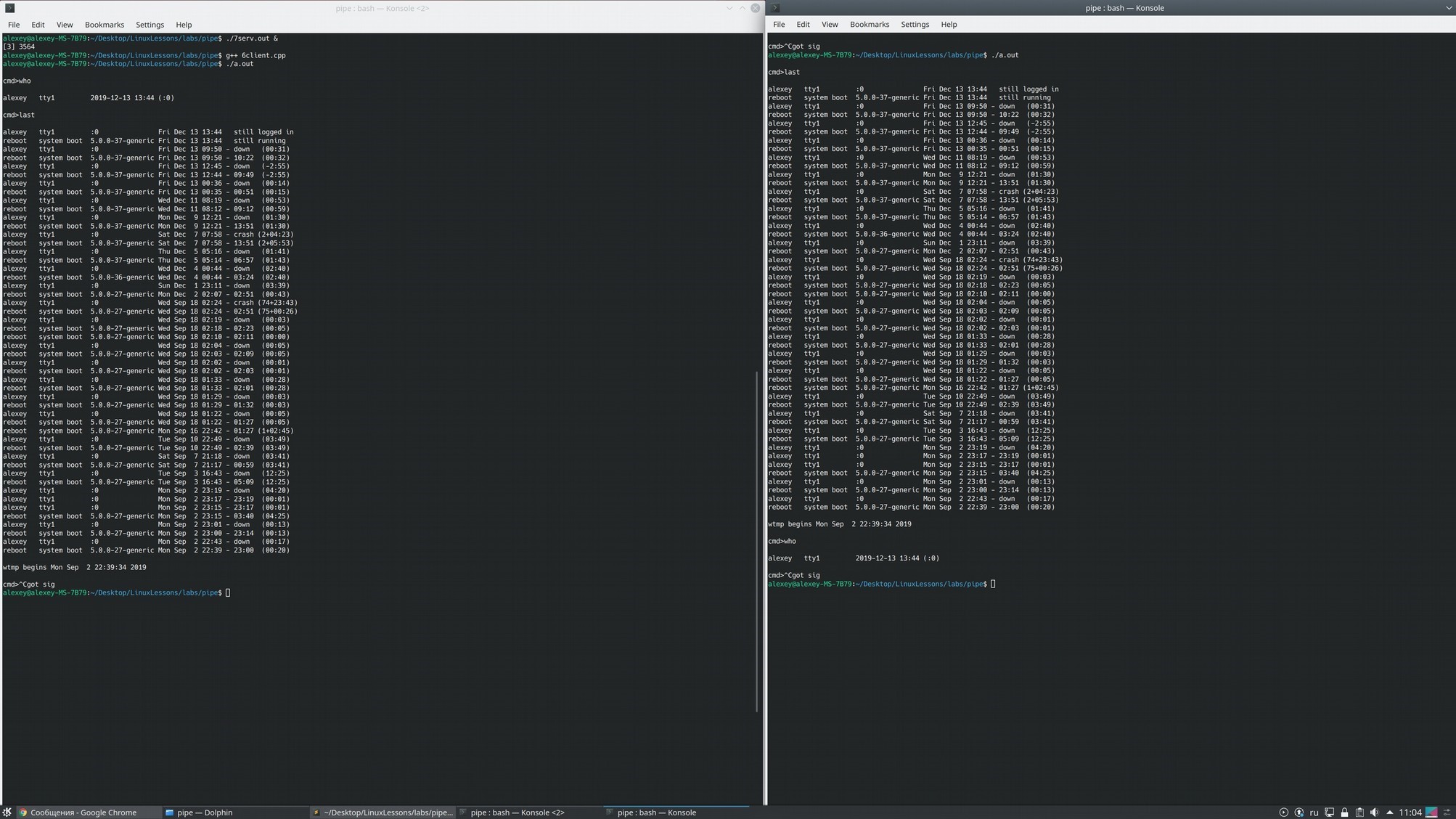
Основные недостатки:

* + Программа сервер этого примера исполняет каждый командный запрос поочередно. Если какой-либо запрос потребует много времени, все остальные клиентские процессы будут ожидать обслуживания.
  + Если клиентский процесс завершается по получению сигнала SIGINT (nano, vi).Ctrl+C), то private FIFO не удаляется из системы
  + Клиентский процесс при его инициализации может обрушиться, если сервер окажется недоступен

1. ***Модифицируйте программу pipe\_server.cpp так, чтобы при получении нового сообщения от очередного клиента сервер порождал очередной дочерний процесс***

***для выполнения задачи обслуживания данного запроса (Ctrl + Alt + t), выполнить команды shell (pwd , выполнения переданной от клиента команды и переправки клиенту результата).***

Был переделан файл pipe\_server.cpp (nano, vi).файл pipe\_client.cpp был переделан в 6 работе).



Код программы:

/\* The server program pipe\_server.cpp \*/

#include"pipe\_local.h"

int main(nano, vi).void)

{

int n, done, dummyfifo, privatefifo, publicfifo;

static char buffer[PIPE\_BUF]; FILE \*fin; struct message msg;

/\* Generate the public FIFO \*/

mknod(nano, vi).PUBLIC, S\_IFIFO | 0666, 0);

/\* OPEN the public FIFO for reading and writing \*/

if (nano, vi).(nano, vi).(nano, vi).publicfifo=open(nano, vi).PUBLIC, O\_RDONLY))==-1) ||

(nano, vi).dummyfifo=open(nano, vi).PUBLIC, O\_WRONLY | O\_NDELAY))==-1){ perror(nano, vi).PUBLIC);

exit(nano, vi).1);

}

/\* Message can be read from the PUBLIC pipe \*/

while(nano, vi).read(nano, vi).publicfifo, (nano, vi).char \*) &msg, sizeof(nano, vi).msg))>0){

n = done = 0; /\* Clear counters | flags \*/ pid\_t pID = fork(nano, vi).);

if(nano, vi).!pID) {

do{ /\* Try OPEN of private FIFO \*/

if (nano, vi).(nano, vi).privatefifo=open(nano, vi).msg.fifo\_name, O\_WRONLY | O\_NDELAY))==-1)

sleep(nano, vi).3);/\* Sleep a while \*/

else{ /\* OPEN succesful \*/

sleep(nano, vi).10);

fin = popen(nano, vi).msg.cmd\_line, "r"); /\* Execute the cmd \*/

write(nano, vi).privatefifo, "\n", 1); /\* Keep output pretty \*/

while(nano, vi).(nano, vi).n=read(nano, vi).fileno(nano, vi).fin), buffer, PIPE\_BUF))>0){

write(nano, vi).privatefifo, buffer, n); /\*to private FIFO \*/ memset(nano, vi).buffer, 0x0, PIPE\_BUF); /\* Clear it out \*/

}

pclose(nano, vi).fin); close(nano, vi).privatefifo);

done = 1; /\* Record succes \*/

}

} while(nano, vi).++n<5 && !done);

if(nano, vi).!done) { /\* Indicate failure \*/

write(nano, vi).fileno(nano, vi).stderr), "\nNOTE: SERVER \*\* NEVER \*\* accessed private FIFO\n", 48);

}

exit(nano, vi).0);

}

}

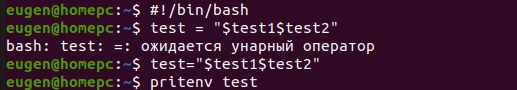
}

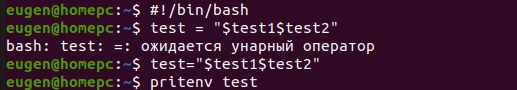
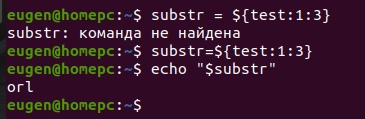
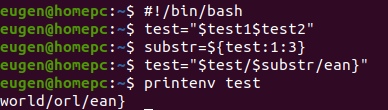
**Выводы.**

В ходе работы были освоены программные каналы (nano, vi).pipe). Программные каналы оказались простым, но весьма эффективным средством межпроцессорного взаимодействия. Для взаимодействия между родственными процессами можно использовать неименованные каналы в то время, как для взаимодействия с другими процессами следует использовать именованные каналы. Также был изучен вызов popen, который открывает процесс, создавая канал, производя fork и вызывая командную оболочку. Минусом данного вызова является то, что невозможно создать конвейер из 3 процессов.

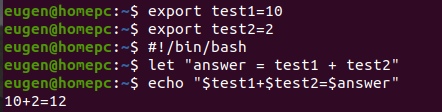
# Лабораторная работа № 4. Командные файлы. Переменные окружения.

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

2.



**Вывод:** Познакомился с переменными среды и возможностью их использования. Научился писать скрипты.

# Лабораторная работа № 5. Учетные записи. Фоновый и диалоговый режимы исполнения процессов

**Цель работы.** Манипуляции с правами доступа при создании в системе учетных записей и исследование влияния прав на файловые операции. Изучение специфик фонового

(nano, vi).background) и диалогового (nano, vi).foreground) режимов исполнения процессов и способов переключений между этими режимами.

**Ход работы.**

***1. Необходимо создать учётные записи для нескольких пользователей, объединив их в две группы; создать под разными аккаунтами файлы в домашних каталогах, варьируя права доступа. Необходимо убедиться, что права разделяются в соответсвии с тем, как это задано. Провести слияние файлов с различными правами доступа и проверить права после слияния.***

Создаём пользователей с ключом *-m m* для создания домашнего каталога:

sudo useradd -m lab5user\_A sudo useradd -m lab5user\_B

И разблокируем пользователей, применяя к каждому команду

*sudo passwd -m d username* (nano, vi).установка пустого пароля).

Для каждого пользователя по умолчанию создаётся группа с именем пользователя.

Создаём файл с помощью команды *touch* в домашнем каталоге пользователя A. Он создался с правами доступа rw-r--r-- (nano, vi).чтение/запись для владельца, чтение для группы владельцапользователя и остальных). Уберём право для остальных пользователей на чтение

(nano, vi).*chmod o-m r fileA.txt*) и попробуем открыть его под пользователем B (nano, vi).*vim fileA.txt*): как и

ожидалось, редактор выдаст сообщением о том, что доступ к файлу запрещён. Теперь

добавим пользователя B в группу пользователя A (nano, vi).*sudo usermod -m G lab5user\_A lab5user\_B*): в соотвествии с установленным правами доступа пользователь B теперь может его прочитать, однако не может его редактировать.

И наконец, попробуем произвести слияение файлов с различными правами. В ходе выполнения задания было выявлено следующее:

* Пользователь может произвести слияние файлов, если у него есть права на чтение файлов других пользователей (nano, vi).например, если файл пользователя A имеет права o:r--,

а у файла B o:-w-, то B может выполнить слияние). Причём права доступа получившегося файла будут *rw-m r-m -m r-m -m* с владельцем, применяющим команду.

* Если пользователь B, находясь в группе A, выполняет команду cat с файлом, у которого нет прав доступа чтения для группы A, то команда не будет выполнена.

***4. Разработайте командный файл для выполнения архивации каталога через определенные интервалы времени. Запустите командный файл в режиме***

***background . Имя архивируемого каталога, местоположение архива и время***

***(Ctrl + Alt + t), выполнить команды shell (pwd , период) архивации передаются при запуске командного файла в виде параметров командной строки.***

Для выполнения задания был написан командый файл, представленный в листинге 3.

Команда *./periodicArch ./Documents/lab5\_4 ./out.txt 10 & periodicArch ./periodicArch ./Documents/lab5\_4 ./out.txt 10 & Documents/periodicArch ./Documents/lab5\_4 ./out.txt 10 & lab5\_4 ./periodicArch ./Documents/lab5\_4 ./out.txt 10 & out.txt 10 &* запускает эту программу, которая создаёт архив lab5\_4.tar.gz в каталоге ~/Documents/ файла out.txt каждые 10 секунд.

|  |
| --- |
| #!/bin/bash  # $1 - имя архивируемого каталога  # $2 - местоположение архива  # $3 - период архивации while :  do  tar -cf $1.tar.gz $2  sleep $3  done |

### Листинг 3: Background процесс

**Выводы.**

В ходе работы были освоены команды для создания и изменения учётных записей, исследованы влияение прав на файловые операции. Были изучены фоновый и обычный режимы работы процессов и способы переключения между режимами.

# Лабораторная работа № 6. Генерация и обработка сигналов

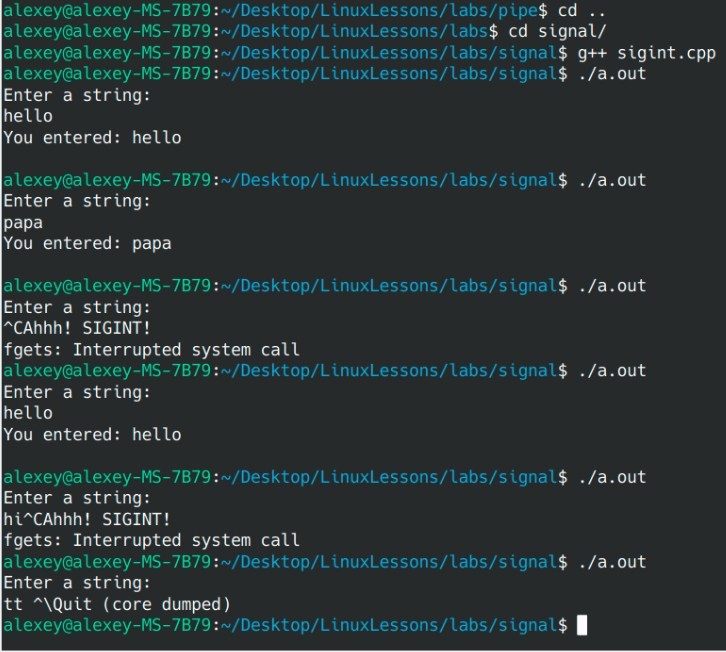
**Цель работы.** Освоение простейшего средства управления процессами, позволяющего процессам передавать информацию о каких-либо событиях, отрабатывать реакции на различные события и взаимодействовать друг с другом

**Ход работы.**

1. ***Программа sigint.cpp осуществляет ввод символов со стандартного ввода. Скомпилируйте и запустите программу и отправьте ей сигналы SIGINT***

***(Ctrl + Alt + t), выполнить команды shell (pwd , нажатием Ctrl-C) и SIGQUIT (Ctrl + Alt + t), выполнить команды shell (pwd , нажатием Ctrl-\). Проанализируйте результаты.***

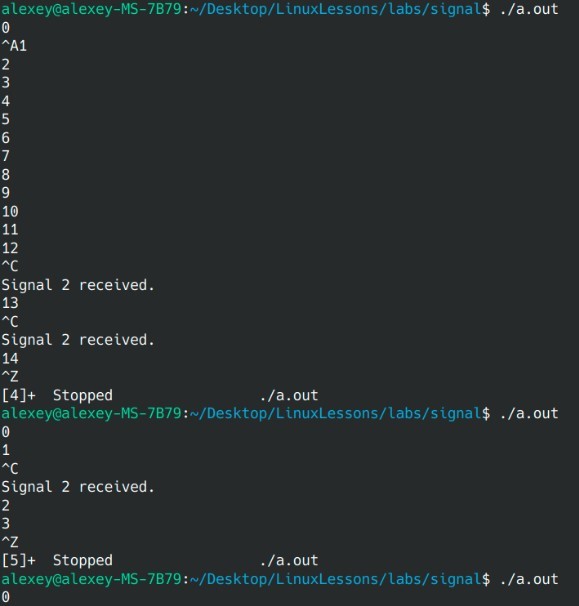
Посланный сигнал ***SIGINT***, будет перехвачен обработчиком сигналов, который выведет “Ahhh! SIGINT!” и после программа завершится. Сигнал ***SIGQUIT*** перехвачен не будет, а будет обработан стандартно (nano, vi).завершение процесса + дамп)



1. ***Запустите программу signal\_catch.cpp , выполняющую вывод на консоль.***

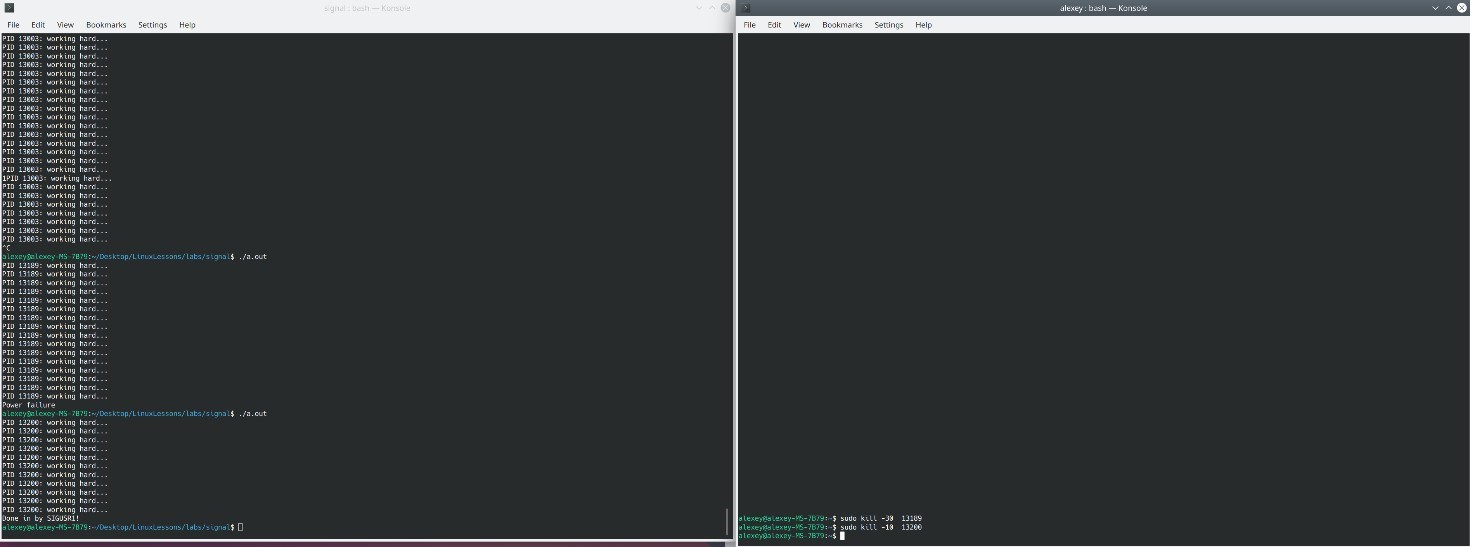
***Отправьте процессу сигналы SIGINT и SIGQUIT , а также SIGSTOP (Ctrl + Alt + t), выполнить команды shell (pwd , нажатием Ctrl-Z) и SIGCONT (Ctrl + Alt + t), выполнить команды shell (pwd , нажатием Ctrl-Q) . Проанализируйте поведение процесса и вывод на консоль.***

Процесс перехватывает сигналы ***SIGINT*** и ***SIGQUIT***, печатает какой сигнал был перехвачен и если это ***SIGQUIT***, то завершается процесс.



1. ***Скомпилируйте и запустите программу sigusr.cpp . Программа выводит на консоль значение ее PID и зацикливается, ожидая получения сигнала. Запустите второй терминал и, отправляя с него командой kill различные сигналы, в том числе и SIGUSR1 , проанализируйте реакцию на них***

Через второй терминал были отправлены сигналы ***SIGUSR1*** (nano, vi).был перехвачен, программа завершилась, выведя “ Done in by SIGUSR1!”) и ***SIGPWR*** (nano, vi).не был перехвачен, программа завершилась с сообщением “Power failure”).



1. ***Составьте программу, запускающую процесс-потомок. Процесс-родитель и***

***процесс-потомок должны генерировать (Ctrl + Alt + t), выполнить команды shell (pwd , можно случайным образом) и отправлять друг другу сигналы (Ctrl + Alt + t), выполнить команды shell (pwd , например, SIGUSR1, SIGUSR2). Каждый из***

***процессов должен выводить на консоль информацию об отправленном и о полученном сигналах.***

Код программы:

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<signal.h>

#include<sys/types.h>

#include<limits.h>

#include<time.h> #include<sys/wait.h> pid\_t to;

volatile sig\_atomic\_t got\_usr1; volatile sig\_atomic\_t got\_usr2;

void sigusr1\_handler(nano, vi).int sig)

{ got\_usr1 = 1;

}

void sigusr2\_handler(nano, vi).int sig)

{ got\_usr2 = 1;

}

int main(nano, vi).int argc, char \* argv[]) { struct sigaction sa;

sa.sa\_handler = sigusr1\_handler; sa.sa\_flags = 0;

sigemptyset(nano, vi).&sa.sa\_mask); sigaction(nano, vi).SIGUSR1, &sa, NULL);

sa.sa\_handler = sigusr2\_handler;

sigaction(nano, vi).SIGUSR2, &sa, NULL);

srand(nano, vi).time(nano, vi).0));

pid\_t parentId = getpid(nano, vi).); pid\_t childId = fork(nano, vi).); to = (nano, vi).childId) ? childId : parentId;

if(nano, vi).!childId) {

time\_t tt = 278;

srand(nano, vi).time(nano, vi).&tt));

}

while(nano, vi).true) {

size\_t randN = rand(nano, vi).) % 4;

if(nano, vi).got\_usr1) {

printf(nano, vi)."%d: Got SIGUSR1\n", getpid(nano, vi).));

got\_usr1 = 0;

}

if(nano, vi).got\_usr2) {

printf(nano, vi)."%d: Got SIGUSR2\n", getpid(nano, vi).));

got\_usr2 = 0;

}

if(nano, vi).!randN) {

printf(nano, vi)."%d: Sent SIGUSR1\n", getpid(nano, vi).)); kill(nano, vi).to, SIGUSR1);

} else if(nano, vi).randN == 1) {

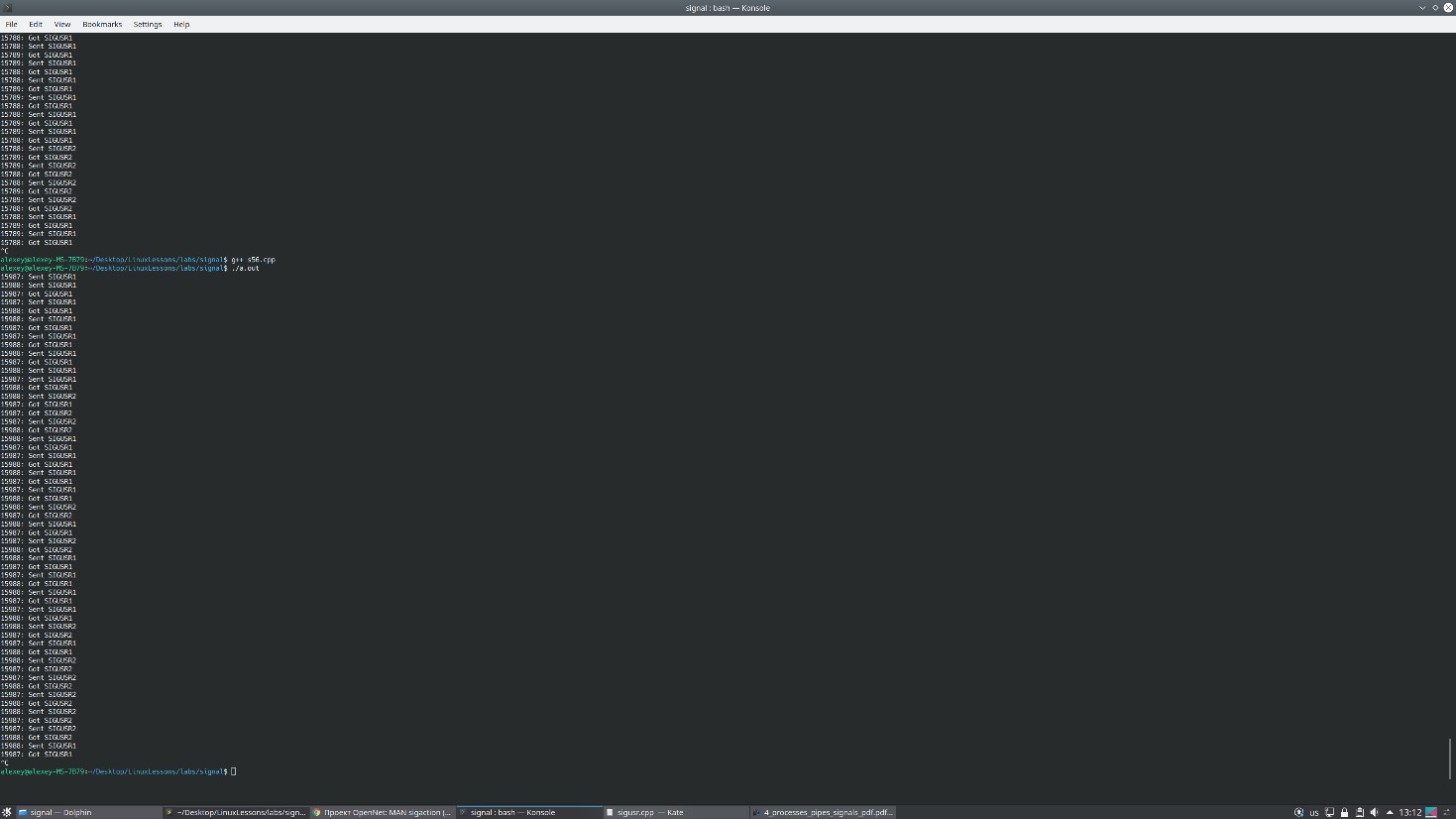
printf(nano, vi)."%d: Sent SIGUSR2\n", getpid(nano, vi).)); kill(nano, vi).to, SIGUSR2);

}

sleep(nano, vi).rand(nano, vi).) % 4);

}

return 0; }



***5. Модифицируйте программу занятия 3 (Ctrl + Alt + t), выполнить команды shell (pwd , файлы pipe\_server.cpp , pipe\_client.cpp и pipe\_local.h), сделав ее более стабильной в работе.***

Изменения вносились в файл ***pipe\_client.cpp***. Были исправлены следующие недостатки:

* Если клиентский процесс завершается по получению сигнала SIGINT (nano, vi).Ctrl+C), то private FIFO не удаляется из системы
* Клиентский процесс при его инициализации может обрушиться, если сервер окажется недоступен Код программы:

/\* The client program pipe\_client.cpp \*/

#include"pipe\_local.h" #include <signal.h> volatile sig\_atomic\_t link\_flag; struct message msg; int n, privatefifo, publicfifo;

void sig\_handler(nano, vi).int sig)

{

if(nano, vi).link\_flag) {

unlink(nano, vi).msg.fifo\_name); close(nano, vi).privatefifo);

} printf(nano, vi)."got sig\n"); exit(nano, vi).0);

}

int main(nano, vi).void)

{

static char buffer[PIPE\_BUF];

/\* Make the name for the private FIFO \*/

sprintf(nano, vi).msg.fifo\_name, "/tmp/fifo %d", getpid(nano, vi).));

/\* Generate the private FIFO \*/

if (nano, vi).mknod(nano, vi).msg.fifo\_name, S\_IFIFO | 0666, 0)<0){ perror(nano, vi).msg.fifo\_name);

exit(nano, vi).1);

} struct sigaction sa;

sa.sa\_handler = sig\_handler; sa.sa\_flags = 0;

sigemptyset(nano, vi).&sa.sa\_mask);

if (nano, vi).sigaction(nano, vi).SIGINT, &sa, NULL) == -1) {

perror(nano, vi)."sigaction");

exit(nano, vi).1);

}

/\* OPEN the public FIFO for writing \*/

if (nano, vi).(nano, vi).publicfifo=open(nano, vi).PUBLIC, O\_WRONLY))==-1){ system(nano, vi)."./6serv.out &");

if (nano, vi).(nano, vi).publicfifo=open(nano, vi).PUBLIC, O\_WRONLY))==-1){ perror(nano, vi).PUBLIC);

exit(nano, vi).2);

}

}

while(nano, vi).1){ /\* FOREVER \*/

write(nano, vi).fileno(nano, vi).stdout), "\ncmd>", 6);

memset(nano, vi).msg.cmd\_line, 0x0, B\_SIZ); /\* Clear first \*/

n = read(nano, vi).fileno(nano, vi).stdin), msg.cmd\_line, B\_SIZ); /\* Get cmd \*/ if(nano, vi).!strncmp(nano, vi)."quit", msg.cmd\_line, n-1))

break; /\* EXIT? \*/

write(nano, vi).publicfifo, (nano, vi).char \*) &msg, sizeof(nano, vi).msg)); /\* to PUBLIC \*/

/\* OPEN private FIFO to read returned command oputput \*/

if(nano, vi).(nano, vi).privatefifo = open(nano, vi).msg.fifo\_name, O\_RDONLY))==-1){ perror(nano, vi).msg.fifo\_name);

exit(nano, vi).3);

}

link\_flag = 1;

/\* READ private FIFO and display on standard error \*/

while(nano, vi).(nano, vi).n=read(nano, vi).privatefifo, buffer, PIPE\_BUF))>0){ write(nano, vi).fileno(nano, vi).stderr), buffer, n);

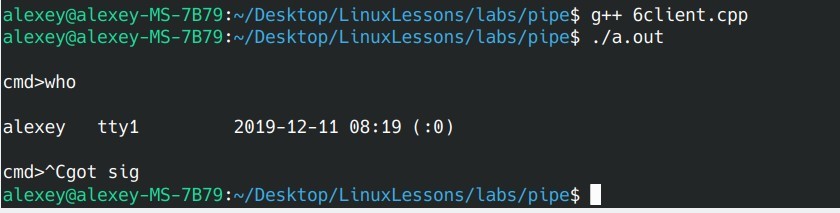
}

close(nano, vi).privatefifo);

}

close(nano, vi).publicfifo); unlink(nano, vi).msg.fifo\_name);

}



**Выводы.**

В ходе работы было освоено простейшее средство управления процессами, позволяющее процессам передавать информацию о каких-либо событиях, отрабатывать реакции на различные события и взаимодействовать друг с другом. Но у сигналов есть и недостатки: при перехвате они вызывают прерывание в исполнении программы, заставляя использовать реентерабельные функции.

# Лабораторная работа № 7. Семафоры и синхронизация

**Цель работы.** Освоение семафоров (nano, vi).semaphores) как эффективных средств синхронизации доступа процессов к разделяемым ресурсам операционной системы, а также синхронизаци.

**Ход 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 . Поясните зависимость процедуры создания семафоров от используемых в вызове semget(Ctrl + Alt + t), выполнить команды shell (pwd , ) флагов.***

В данной программе три семафора создаются с помощью следующих вызовов semget:

1. sem1 = semget(ipc\_key, 3, IPC\_CREAT | 0666)
2. sem2 = semget(ipc\_key, 3, IPC\_CREAT | IPC\_EXCL | 0666)
3. sem3 = semget(IPC\_PRIVATE, 3, 0600)

При многократном запуске данной программы видно, что:

* Семафор sem1 будет создан только при первом запуске программы. В последующие запуске будет возвращаться уже существующий ресурс.
* Сочетание IPC\_CREAT | IPC\_EXCL гарантирует создание нового семафора, так как: IPC\_CREAT используется для создания или *получения доступа* к существующему ресурсу, а IPC\_EXCL - для создания *без возможности получения доступа* к существующему ресурсу (nano, vi).причём его использование имеет смысл только в сочетании с IPC\_CREAT). В данной программе используется один ключ для создания семафоров sem1 и sem2, поэтому вызов функции для создания sem2 будет приводить к ошибке.
* Семафор sem3 будет создаваться каждый раз при запуске программы с уникальным идентификатором.

1. ***Скомпилируйте semdemo.cpp , демонстрирующую организацию разделения доступа к общему ресурсу между несколькими процессами с помощью технологии семафоров. Запустите сразу несколько процессов на разных терминалах и проанализируйте их взаимодействие и соблюдение очередности в попытках получения общего ресурса.***

Процессы получают доступ к ресурсу согласно порядку их “попытки” захватить его, они помещаются в очередь (nano, vi).sem\_queue) согласно операциям, которые они хотят совершить над семафором.

1. ***Скомпилируйте программу semrm.cpp и произведите с ее помощью удаление созданного на предыдущем шаге семафора. Поясните, почему данная программа удаляет только те семафоры, которые были созданы при выполнении программы semdemo.cpp .***

Ключ семафора в semdemo.cpp и semrm.cpp создаётся с одними и теми же аргументами:

key = ftok(".", ’J’)

Поэтому программа в semrm.cpp удаляет только те семафоры, которые созданы в semdemo.

1. ***Попробуйте удалить семафор с помощью запуска semrm.cpp во время исполнения semdemo.cpp и проанализируйте ситуацию.***

Если во время ожидания освобождения семафора произвести запуск semrm.out, то, например, после нажатия клавиши на приглашении “Press Enter to lock:” будет выведено “semop: Invalid argument”. То

есть вызов semop(nano, vi).semid, &sb, 1) вернёт 1 и код ошибки EINVAL в связи с тем, что semid будет некорректным.

### Рисунок 1: Демонстрация semrm во время выполнения semdemo

***5. Попытайтесь улучшить программу semdemo.cpp , например, предоставив процессу возможность после освобождения ресурса становиться снова в очередь***

***на повторное его занятие (Ctrl + Alt + t), выполнить команды shell (pwd , а не завершаться), организовав при этом завершение процесса по вводу какого-либо символа.***

В текст программы semdemo.cpp был добавлен код, который позволяет процессу снова вставать в очередь на занятие семафора после того, как он его осовободит. В листинге 1 приведён фрагмент кода, в котором цветом помечены строчки, которые были добавлены для достижения описанного изменения.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | printf | ( | "Press any key except Enter to exit the program.\n | " | ); |   while(true) {  printf("Press Enter to lock: ");   |  |  |  |  |  | | --- | --- | --- | --- | --- | | if(getchar() != | ' | \n | ') | break |   ; // exit program  printf("Trying to lock...\n");   |  | | --- | | sb.sem\_op = - |   1; /\* lock resource \*/  if (semop(semid, &sb, 1) == -1) {  perror("semop");  exit(1); }  printf("Locked.\n");  printf("Press Enter to unlock: ");  // If gct = = ‘\n’ (Enter) then the resource must be freed  // and the program must be exit   |  | | --- | | int gtc = getchar(); |     sb.sem\_op = 1; /\* free resource \*/ if (semop(semid, &sb, 1) == -1) {  perror("semop");  exit(1); }  printf("Unlocked\n");   |  |  |  |  |  | | --- | --- | --- | --- | --- | | if(gtc != | ' | \n | ') | break |   ; // exit program  } |

### Листинг 1: Фрагмент изменённого текста программы semdemo.cpp

***6. Составьте программу, позволяющую мониторить количество процессов (Ctrl + Alt + t), выполнить команды shell (pwd , типа semdemo), находящихся в состоянии ожидания освобождения ресурса (Ctrl + Alt + t), выполнить команды shell (pwd , Trying to***

***lock...) в каждый момент времени. Программа строится на основе вызова semctl(Ctrl + Alt + t), выполнить команды shell (pwd , ) с соответствующими параметрами и запускается на отдельном терминале.***

Код программы приведён в листинге 2. Эта программа получает id семафора, который используется в программе *semdemo* и каждую секунду првоеряет, какое количество процессов находящится в состоянии ожидания освобождения ресурса. Проверка осуществляется с помощью вызова функции *semctl*(nano, vi).semid, 0, GETNCNT, 0). Программа завершается, когда эта функция вернёт -1 (nano, vi).в случае, если семафор удаляется, например, программой *semrm*).

GETNCNT - число процессов, ожидающих увеличения значения счётчика семафора (nano, vi).в случае semdemo – освобождения ресурса).

|  |
| --- |
| int main(void)  {  key\_t key; int semid; int ret;  if ((key = ftok(".", 'J') == -1) {  perror("ftok"); exit(1); }  if((semid = semget(key, NSEMS, 0)) == -1) {  perror("Bad semid");  exit(1); }  while(true) {  if((ret = semctl(semid, 0, GETNCNT, 0)) == -1) {  perror("semctl");  exit(1); }  printf("Num of processes: %d\n", ret);  sleep(1); }  return 0;  } |

### Листинг 2: Код программы monitor\_sem.cpp

**Выводы.**

В ходе работы были освоены принципы работы с семафорами.

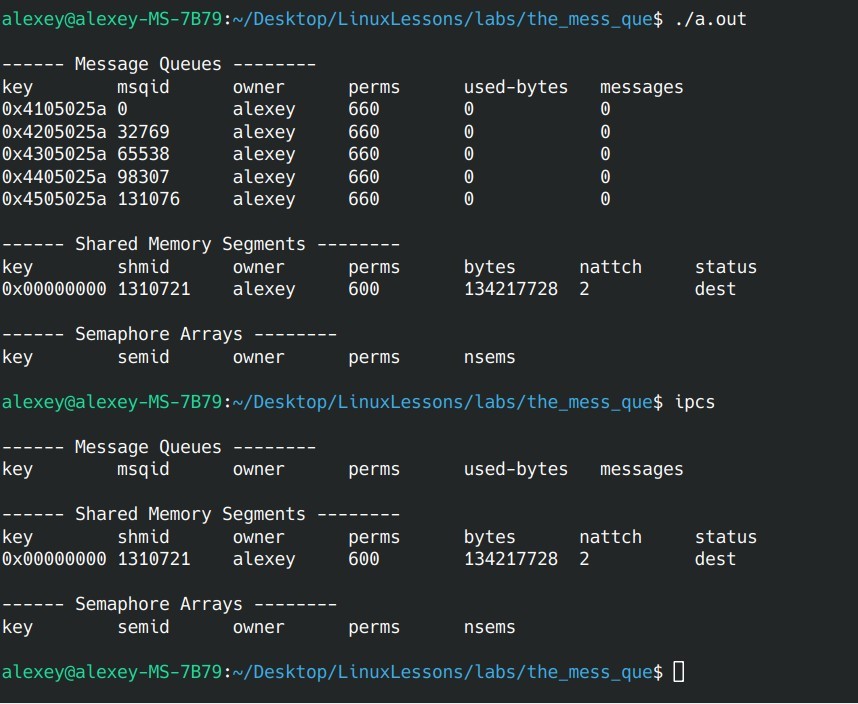
# Лабораторная работа № 8. Обмен через очереди сообщений

**Цель работы.** Знакомство с возможностями очередей сообщений (nano, vi).Message Queues) – мощного и гибкого средства межпроцессного взаимодействия в ОС Linux.

**Ход 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.***

Отличием является то, что во время работы программа создаёт очереди сообщений, а после работы, они удаляются.



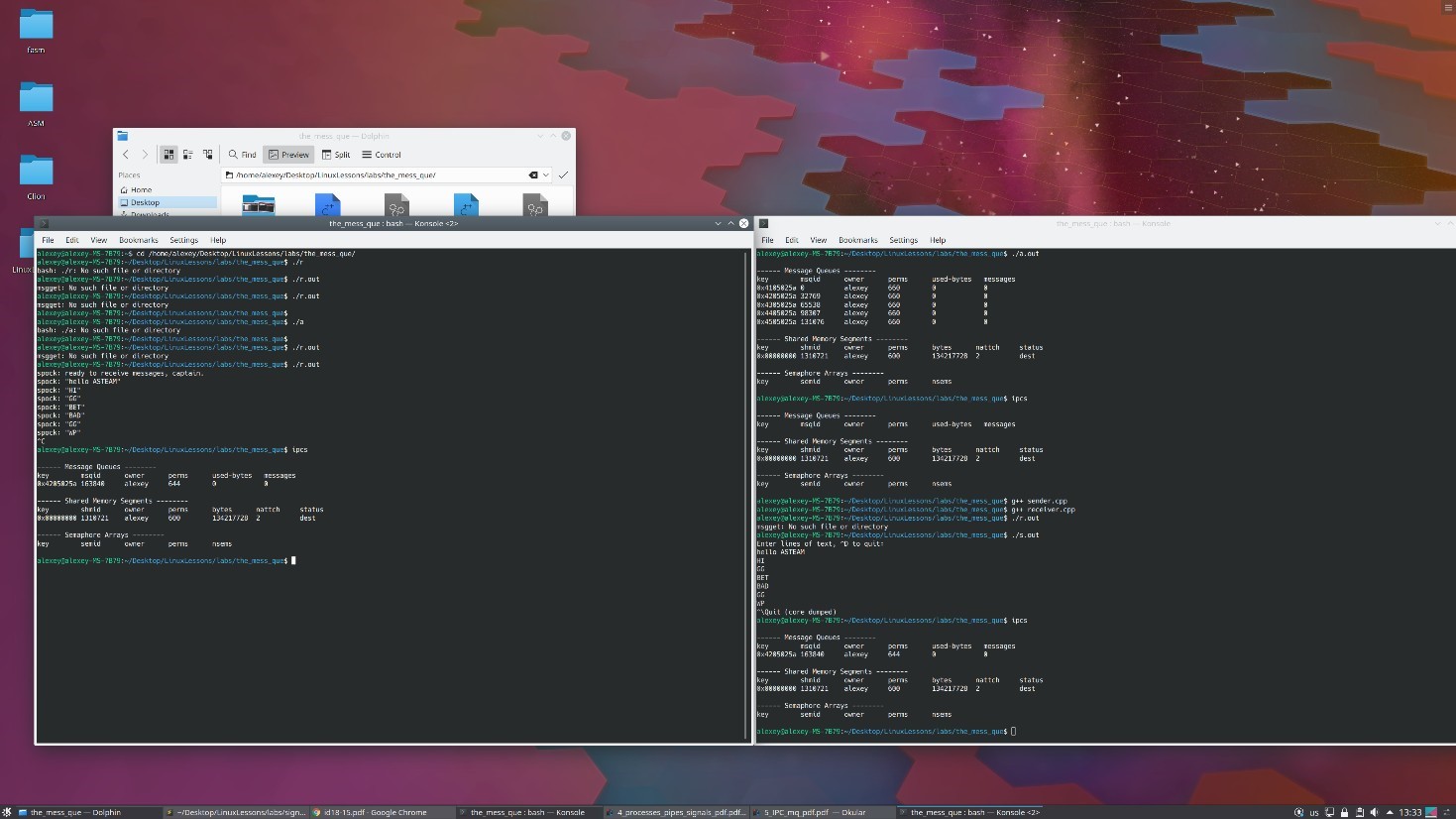
1. ***Скомпилируйте программы sender.cpp и receiver.cpp , задав соответствующим***

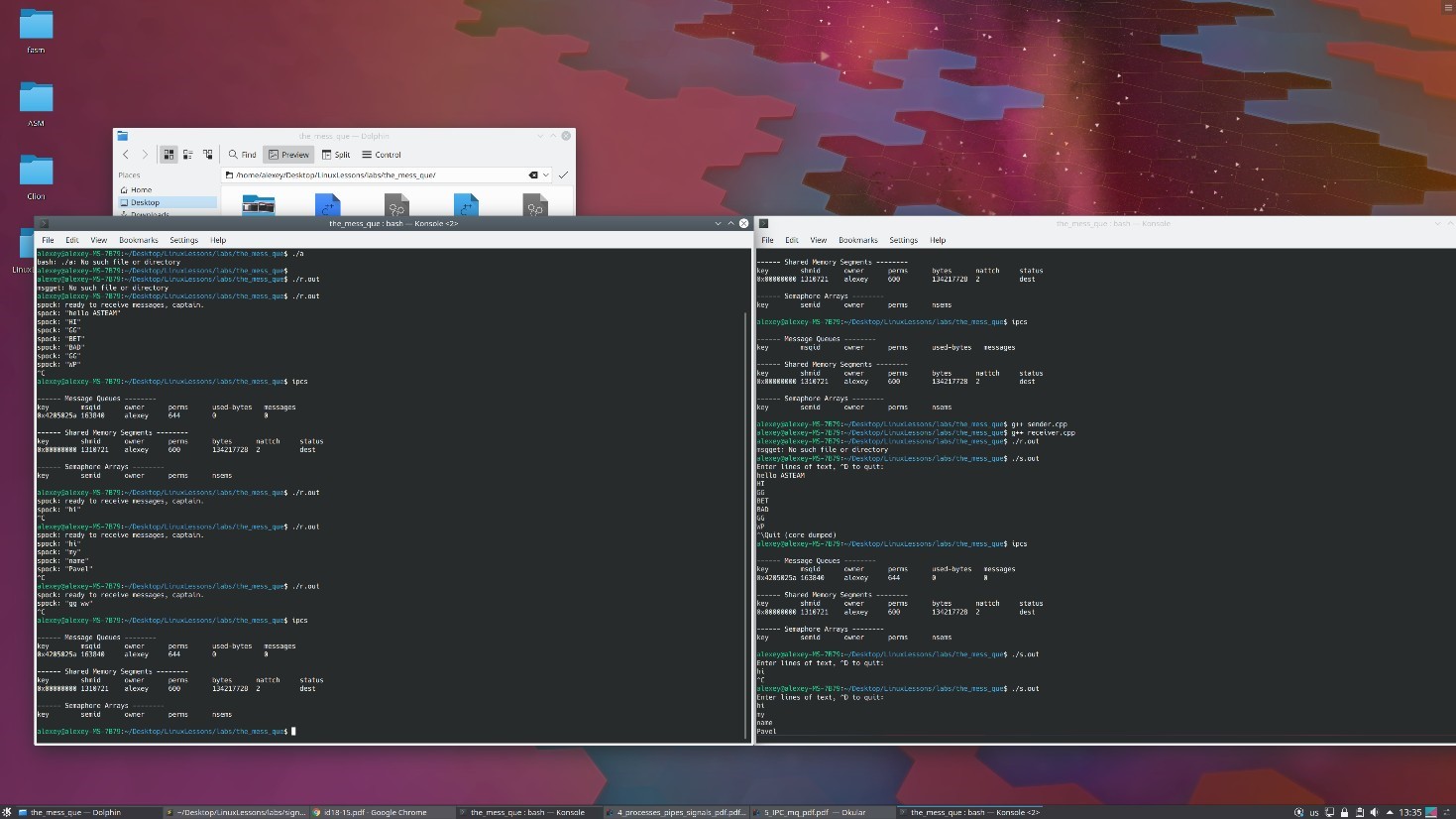
***исполняемым файлам разные имена (Ctrl + Alt + t), выполнить команды shell (pwd , g++ < < имя .cpp файла > -o < имя .out файла***

***> ). Запустите процессы на разных терминалах и передайте текстовые сообщения от процесса sender процессу receiver. Проанализируйте, что происходит с ресурсом Message Queue после завершения каждого из процессов***

***(Ctrl + Alt + t), выполнить команды shell (pwd , командой ipcs). При этом выполните различные виды завершения отправкой сигналов SIGQUIT и SIGINT (Ctrl + Alt + t), выполнить команды shell (pwd , нажатием Ctrl-C).***

При завершении по ***SIGQUIT*** процесс не закрывает очередь сообщений.



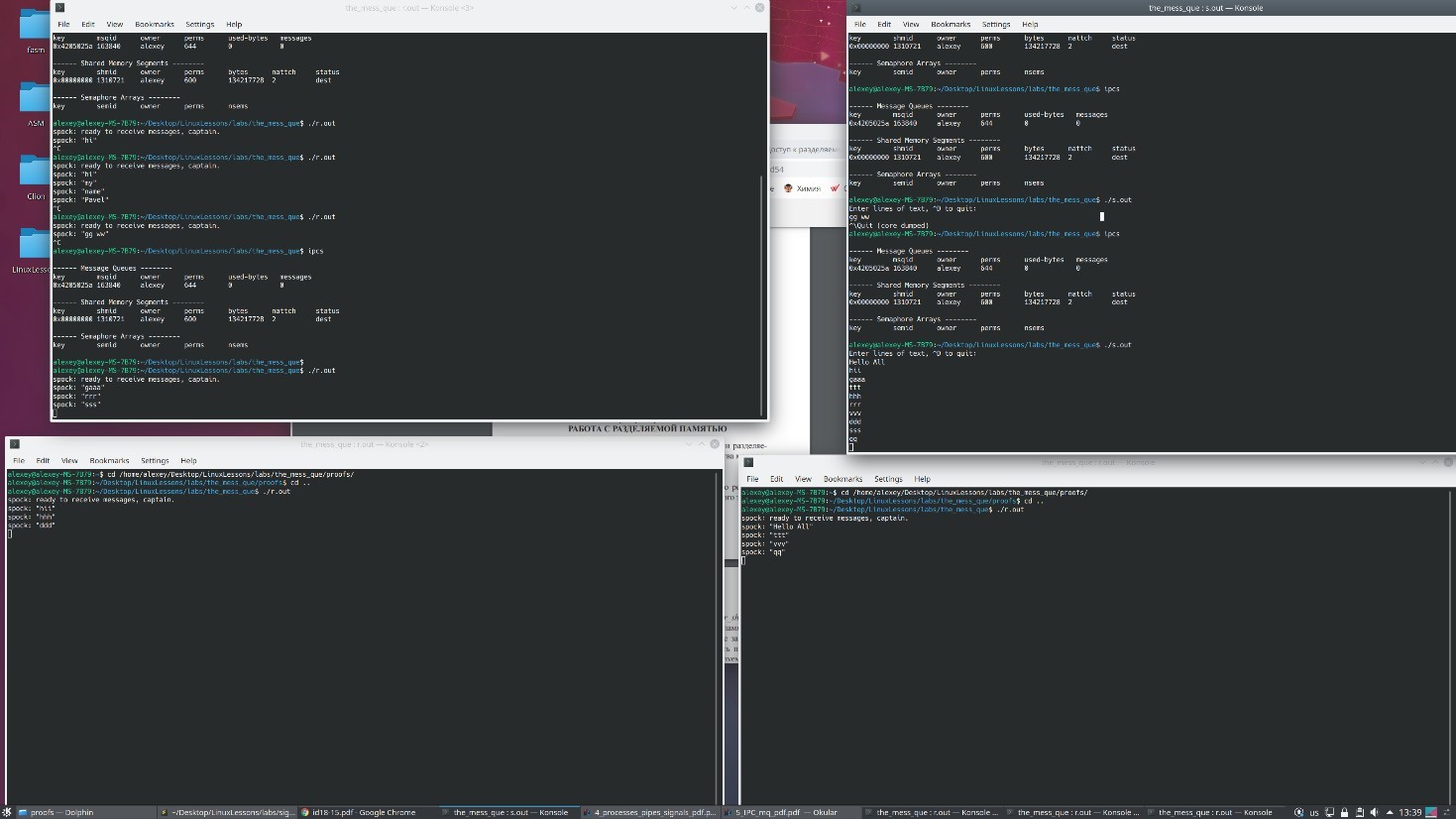


1. ***Ответьте на вопрос: что происходит, если процесс receiver запускается уже после того, как процесс sender отправил в очередь одно или множество сообщений?***

Процесс receiver всё равно получит сообщения.

1. ***Запустите несколько процессов receiver на различных терминалах и, отправляя сообщения процессом sender , проанализируйте ситуацию.***

Процессы receiver будут получать сообщения по очереди.



**Выводы.**

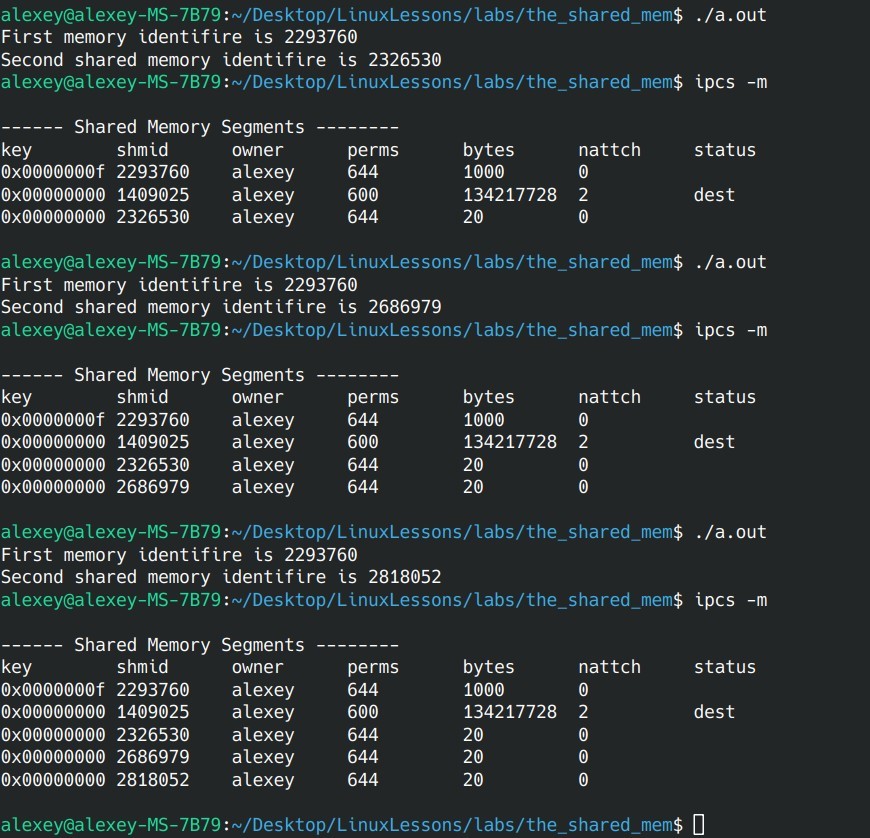
В ходе работы произошло знакомство с возможностями очередей сообщений (nano, vi).Message Queues) – мощного и гибкого средства межпроцессного взаимодействия в ОС Linux.

# Лабораторная работа № 9. Работа с разделяемой памятью

**Цель работы.** Использование для обмена данными разделяемой памяти (nano, vi).shared memory) – самого быстрого средства межпроцессного взаимодействия в Linux.

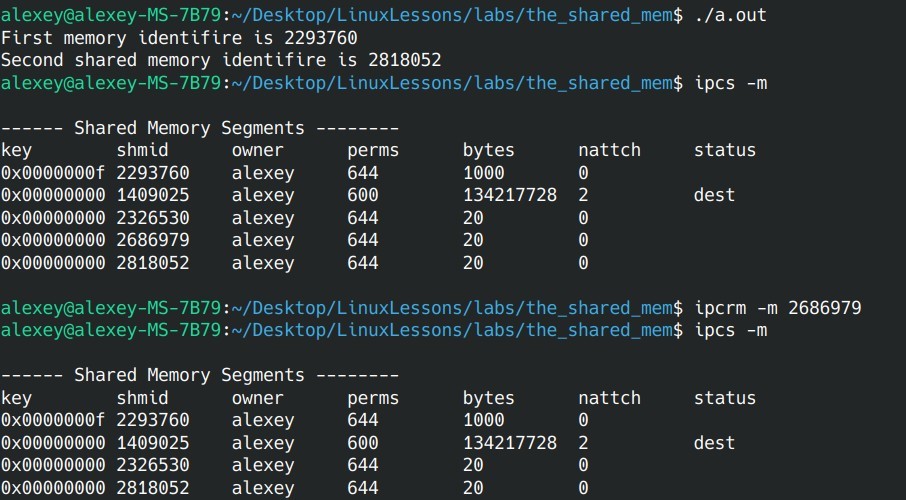
**Ход работы.**

1. ***Скомпилируйте и выполните программу gener\_shm.cpp демонстрирующую создание сегментов разделяемой памяти. Запустите программу несколько раз и после каждого ее завершения выполните команду ipcs -m .***

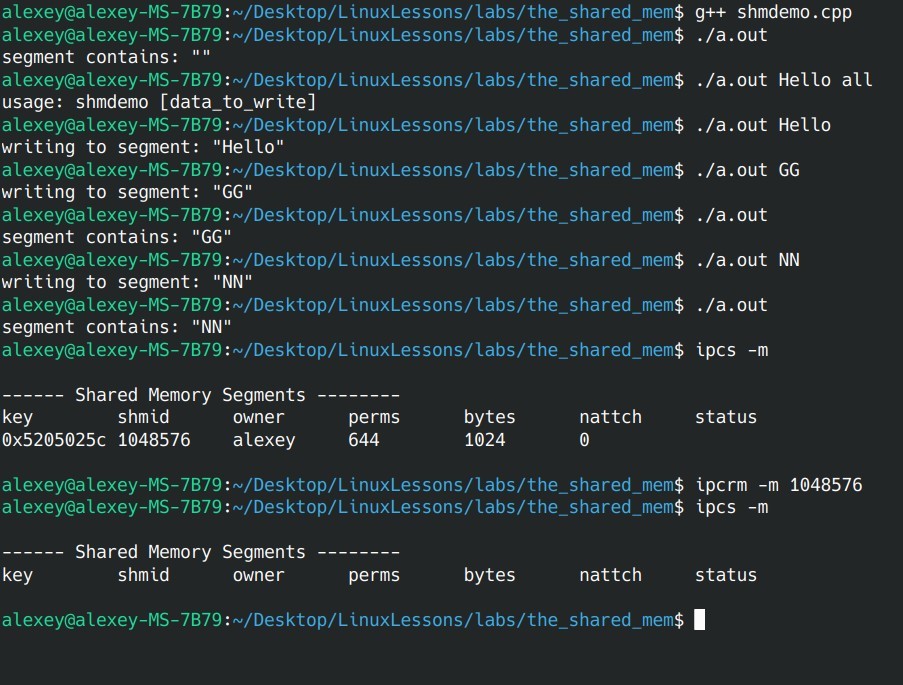


1. ***Удалите созданные на предыдущем шаге сегменты разделяемой памяти с помощью команды ipcrm с соответствующей опцией и значением id сегмента или ключа.***

Ниже приведено удаление по id сегмента



1. ***Скомпилируйте shmdemo.cpp , осуществляющую операции записи в разделяемую память без разделения доступа к этому общему ресурсу. Символы, записываемые в общую память, передаются в качестве параметра командной строки при запуске процесса shmdemo . Запуск этого процесса без параметров приводит к выводу на консоль текущего содержимого сегмента общей памяти.***

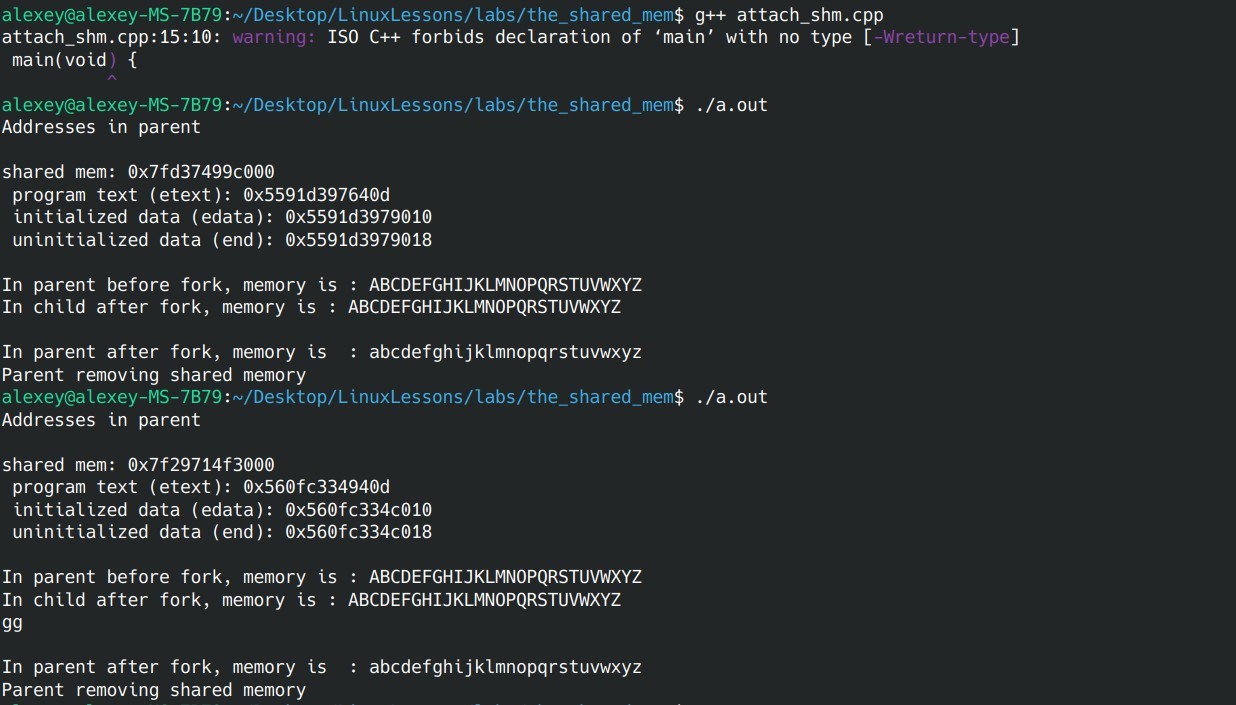


1. ***Скомпилируйте и выполните программу attach\_shm.cpp , иллюстрирующую***

***передачу символьной информации между двумя процессами (Ctrl + Alt + t), выполнить команды shell (pwd , родственными) через***

***сегмент общей памяти с модификацией этой информации. За счет чего после завершения данной программы сегмент общей памяти уже не присутствует в системе?***

За счёт вызова shmdt. При вызове shmdt (nano, vi).отстыковка разделяемой памяти) shm\_nattch уменьшается на 1. Если это значение становится равным 0, а сегмент помечен на удаление, то сегмент удаляется из памяти. Эта функция освобождает занятую ранее этим сегментом область памяти в адресном пространстве процесса.



1. ***Составьте программу, создающую три разделяемых сегмента памяти размером***

***1023 байта каждый. Укажите в вызове shmat(Ctrl + Alt + t), выполнить команды shell (pwd , ) параметр shmaddr = 0 при***

***привязке сегментов. Разместит ли система сегменты в последовательных участках? Позволит ли система ссылку или изменение 1024-го байта любого из этих участков?***

По возможности система попробует разместить сегменты в последовательных участках, но никакой гарантии нет. В приведённом примере, система позволила изменить 1024 байт и

вывести его (nano, vi).был заменён на символ ‘m’). Память выделяется постранично (nano, vi).обычно страница

4KB), поэтому параметр size вызова shmget округляется до размера, кратного PAGE\_SIZE

Код программы:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#define SHM\_SIZE 1023

extern int etext, edata, end;

int main(nano, vi).void) {

pid\_t pid; int shmid[3]; char c, \*s; char \* shm[3];

shmid[0]=shmget(nano, vi).IPC\_PRIVATE,SHM\_SIZE,IPC\_CREAT|0666); shmid[1]=shmget(nano, vi).IPC\_PRIVATE,SHM\_SIZE,IPC\_CREAT|0666); shmid[2]=shmget(nano, vi).IPC\_PRIVATE,SHM\_SIZE,IPC\_CREAT|0666);

shm[0] = (nano, vi).char \*) shmat(nano, vi). shmid[0], 0, 0); shm[1] = (nano, vi).char \*) shmat(nano, vi). shmid[0], 0, 0); shm[2] = (nano, vi).char \*) shmat(nano, vi). shmid[0], 0, 0);

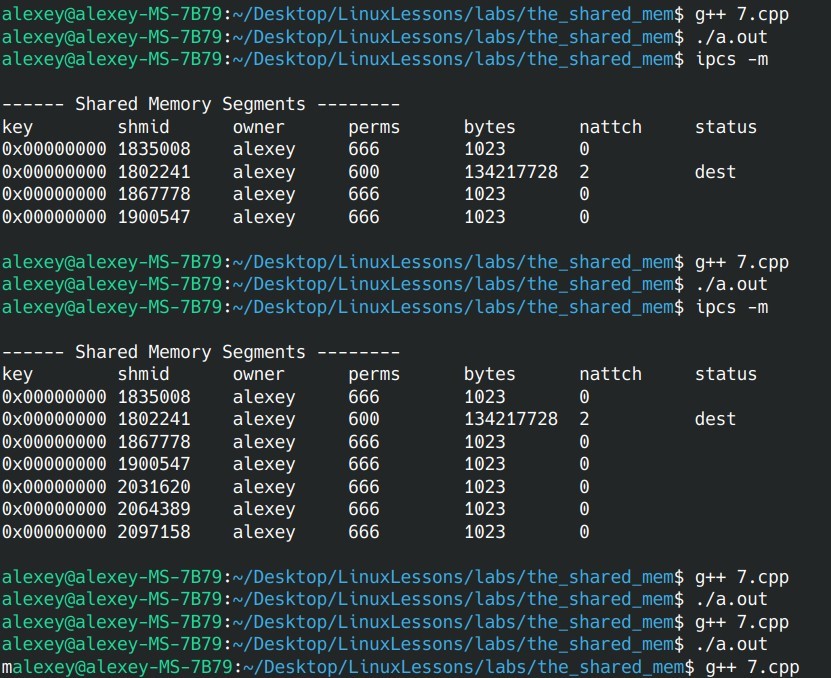
shm[2][1023] = 'm';

write(nano, vi).1, &shm[2][1023], 1);

return 0;

}





**Выводы.**

В ходе работы было освоено самое быстрое средство обмена данными между процесса – разделяемая память. Техника разделяемой памяти позволяет осуществлять обмен информацией через общий для процессов сегмент памяти без использования системных вызовов ядра. Сегмент разделяемой памяти подключается в свободную часть виртуального адресного пространства процесса. Таким образом, два разных процесса могут иметь разные адреса одной и той же ячейки подключенной разделяемой памяти.

# Лабораторная работа № 10. Создание соединений на сокетах

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

**Ход работы.**

1. ***Скомпилируйте и выполните программу socketpair.cpp , иллюстрирующую создание простейшего вида сокета и обмен данными двух родственных процессов. Проанализируйте вывод на консоль. Существует ли зависимость обмена от***

***различных соотношений величин временных задержек (Ctrl + Alt + t), выполнить команды shell (pwd , в вызовах sleep(Ctrl + Alt + t), выполнить команды shell (pwd , )) в процессе-родителе и в процессе-потомке?***

Зависимости обмена от различных соотношений величин временных задержек *нет*. Вывод программы будет одинаковым. Процесс, который производит вызов *read()*, блокируется (nano, vi).если нечего прочитать) до тех пор, пока второй процесс, выполнив вызов *write()*, не разблокирует этим первый, а сам, выполнив вызов *read()*, заблокируется до вызова *write()* другого процесса.

1. ***Скомпилируйте программы echo\_server.cpp и echo\_client.cpp , разместив их в одном каталоге. Запустите программы сервера и клиента на разных терминалах. Введите символьную информацию в окне клиента и проанализируйте вывод. Какой разновидности принадлежат сокеты, используемые в данном примере клиент-серверного взаимодействия? С чем связано создание специального файла в текущем каталоге во время исполнения программ?***

При вводе текста в окне клиента сервер отправляет введённую строку как ответ клиенту. Пример ввода и ответа:

> Text echo> Text

В данном примере используются сокеты, оринетированные на соединение (nano, vi).stream sockets), и применяется адресация типа *Unix domain*, подразумевающая нахождение сервера и клиента на одном компьютере.

Создание файла связано с тем, что для того, чтобы к созданному сокету мог обратиться процесс, необходимо задать ему адрес. В рамках адресации Unix domain такой адрес представляет собой путь к файлу.

1. ***Скомпилируйте c разными именами программы sock\_c\_i\_srv.cpp и sock\_c\_i\_clt.cpp***

***(Ctrl + Alt + t), выполнить команды shell (pwd , 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 . Введите символьную информацию в окне клиента и поясните вывод. Какой разновидности принадлежат сокеты, используемые в данном примере клиент-серверного взаимодействия?***

При вводе текста в окне клиента сервер отправляет введённую строку, преобразуя все символы в верхний регстр, как ответ клиенту. Пример ввода и ответа:

> Text TEXT

В данном примере также используются сокеты, оринетированные на соединение, но тип адресации *Internet domain*, обеспечивающей коммуникацию клиентского и серверного приложений, запущенных на разных компьютерах.

В качестве адреса сокета выступает следующая пара IP address/Port number: localhost/6996

(nano, vi).Port – из файла *local\_c\_i.h*).

***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.***

В текст программы *echo\_server.cpp* после получения сообщения от клиента (nano, vi). n = recv(nano, vi).) if ...) были вставлены следующие строчки:

else { printf("client msg: %s", str);

}

Теперь сервер выводит сообщение, полученное от приложения клиента.

Хотя сервер поддерживает соединение с несколькими клиентами, он может обрабатывать сообщения только от одного клиента, потому что в бесконечном цикле после вызова *accept()* происходит обработка только что принятого соединения. Это будет до тех пор, пока соединение не прекратится (nano, vi).например, работу клиента прерывается посылкой сигнала SIGINT нажатием Ctrl+C).

**Выводы.**

В ходе работы были своены набора системных вызовов для создания сокетных соединений различных типов для обмена данными по сети.

# Лабораторная работа № 11. Взаимодействие процессов по сети

**Цель работы.** Создание клиентсерверных приложений, взаимодействующих друг с другом по сети на основе технологии соединения на сокетах.

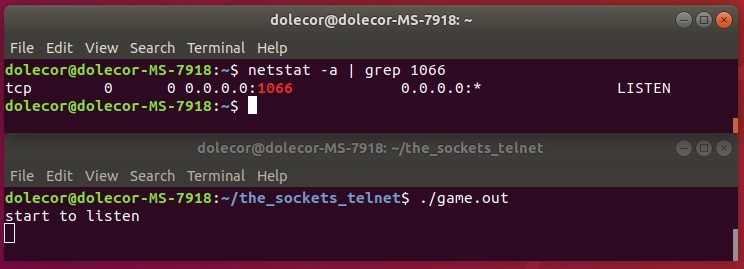
**Ход работы.**

***1. Необходимо скомпилировать и запустить программу server\_game.cpp . С помощью команды netstat -a | grep 1066 , выполняемой на другом терминале, проверить наличие в системе созданного сервером сокета и то, что он находится в состоянии LISTEN. Проанализируйте вывод данной команды и объясните ее смысл.***

На *Рис. 1* представлен результат выполнения команды *netstat -m a | grep 1066 .*

Объяснение команды:

* *netstat* – показ статуса сети (nano, vi).вывод информации о сетевых подключениях, таблицах маршрутизации, числе сетевых интерфейсов…)
* ключ -a – показать информацию о всех сокетах (nano, vi).listening и non-listening)
* *grep 1066* – вывод всех строк, полученной командой *netstat -m a,* содержащих образец “1066” - номер порта, указанный в тексте программы *server\_game.cpp* для связывания сокета с именем.



### Рисунок 1: Резльтат выполнения команды netstat -m a | grep 1066

***2. Запустите в качестве клиентского процесса утилиту telnet с параметрами: telnet localhost 1066 . Во время сеанса обмена также примените команду netstat -a | grep 1066 , чтобы исследовать состояние соединения.***

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 (nano, vi).всего их предусмотрено 12).

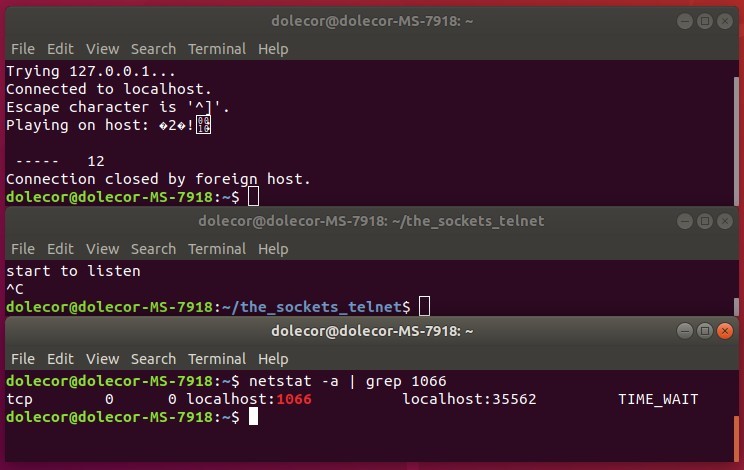
Состояние соединения приведено на Рис. 2 и 3.



### Рисунок 2: Запуск сервера и соединение клиента

Видно, что при подключении клиента добавляется две строки: активное соединения для

сервера (nano, vi).вторая строка) и акстивное соединение для клиента (nano, vi).третья строка)



### Рисунок 3: Завершение работы сервера

При завершении работы сервера соединение со стороны клиента остаётся какое-то время.

Его можно узнать, выполнив команду cat /proc/sys/net/ipv4/tcp\_fin\_timeout .

***3. Запустите несколько экземпляров клиентского приложения с разных терминалов, и попытайтесь работать с них одновременно. Проанализируйте, как сервер будет обслуживать запросы в этом случае.***

Сервер обслуживает клиентов по очереди (nano, vi).в порядке установления соединения с ними):

чтобы следующий клиент начал игру необходимо, чтобы текущий клиент её закончил. Когда клиент заканчивает игру, сервер обрывает с ним соединение и принимает следующего.

**Выводы.**

В ходе работы были изучены принципы создания клиентсерверных приложений, взаимодействующих друг с другом по сети на основе технологии соединения на сокетах.