Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

**Высшая школа интеллектуальных систем и суперкомпьютерных технологий.**

**Отчеты по лабораторным работам**

по дисциплине «Системное программное обеспечение GNU/Linux»

Выполнили

студенты гр. 3530901/90203 В.Д. Коротких, А.А. Ундольский

Руководитель

Старший преподаватель З. В. Куляшова

Санкт-Петербург 2020

Оглавление

[**Лабораторная работа №1 «Базовые команды ОС»** 3](#_Toc88770863)

[**Лабораторная работа №2 «Работа с процессами в OS Linux»** 9](#_Toc88770864)

[**Лабораторная работа №3 «Программные каналы»** 24](#_Toc88770865)

[**Лабораторная работа №4 «Командные файлы. Переменные окружения»** 35](#_Toc88770866)

[**Лабораторная работа №5 «Учетные записи. Фоновый и диалоговый режимы исполнения процессов»** 40](#_Toc88770867)

[**Лабораторная работа №6 «Генерация и обработка сигналов»** 47](#_Toc88770868)

[**Лабораторная работа №7 «Семафоры и синхронизация»** 55](#_Toc88770869)

[**Лабораторная работа №8 «Обмен через очереди сообщений»** 66](#_Toc88770870)

[**Лабораторная работа №9 «Работа с разделяемой памятью»** 78](#_Toc88770871)

[**Лабораторная работа №10 «Создание соединений на сокетах»** 85](#_Toc88770872)

# **Лабораторная работа №1 «Базовые команды ОС»**

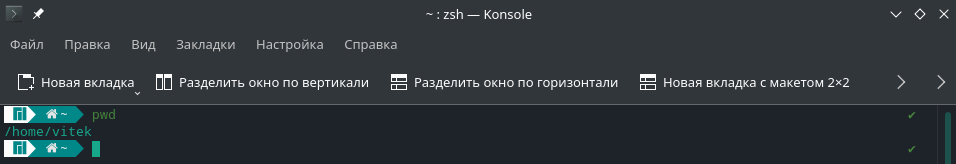
**Цель работы**

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

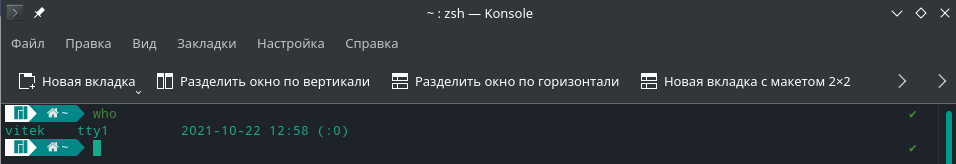
**Последовательность выполнения работы:**

1. Войдите в систему под логином вашей учебной группы, получив необходимый пароль у преподавателя.
2. Запустите терминал нажатием комбинации клавиш Ctrl + Alt + t
3. Выполните на терминале команды shell, рассмотренные в материалах лекций. Такие, как pwd, who, ls, cd, mkdir, rm, chmod. Полное описание синтаксиса и семантики этих и любых других команд можно увидеть в системе помощи ОС Linux, вызываемой с терминала в виде man <интересующая вас команда>, или запускайте веб-браузер и используйте всю информационную мощь Интернета.
4. Проанализируйте результаты выполнения команд. Наиболее значимые скриншоты (снимаются нажатием клавиш Alt + Prnt Scrn) поместите в отчет.

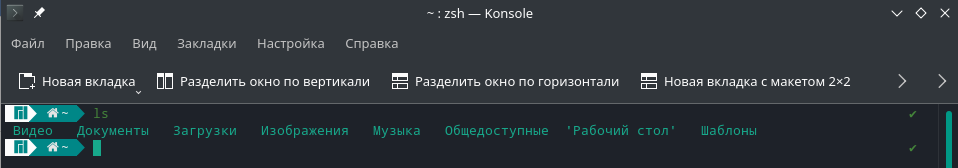
pwd – вывод на экран имени текущей/рабочей директории



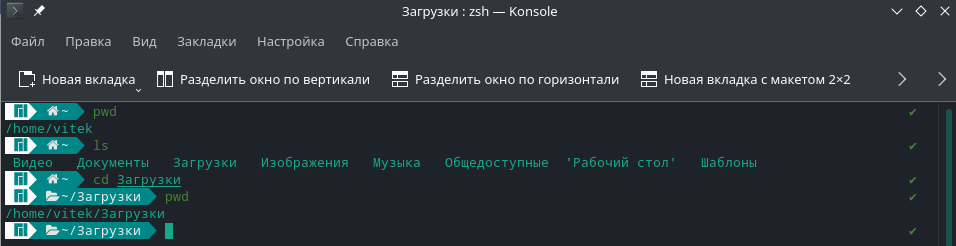
who – показывает информацию о пользователях, вошедших в систему



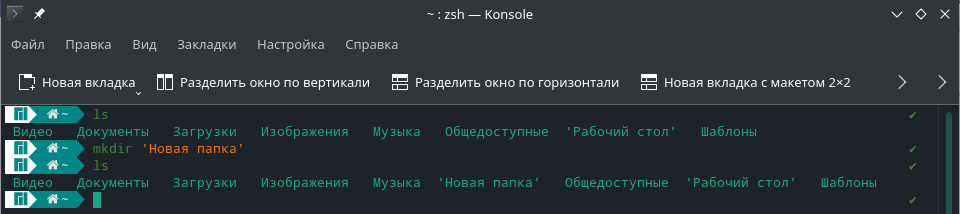
ls – выдаёт информацию о файлах (по умолчанию о текущем каталоге)



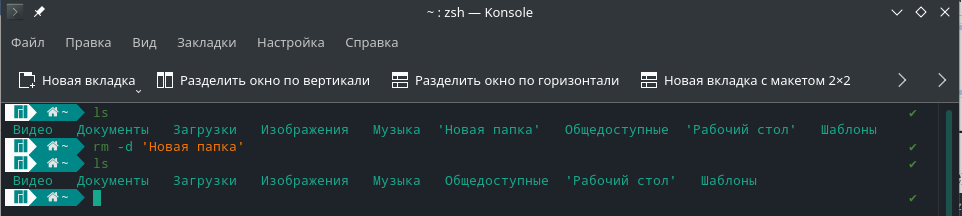
cd – сменить текущую директорию на другую



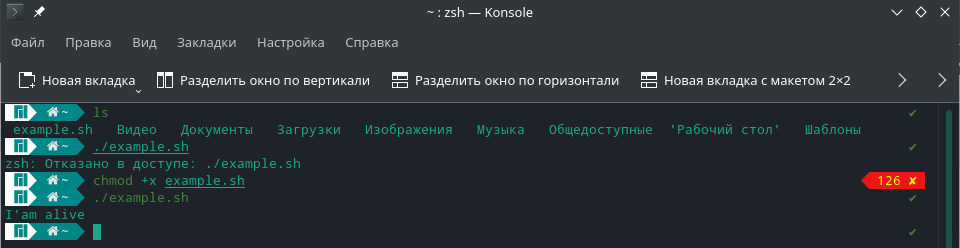
mkdir – cоздаёт каталог(и), если он ещё не существует.



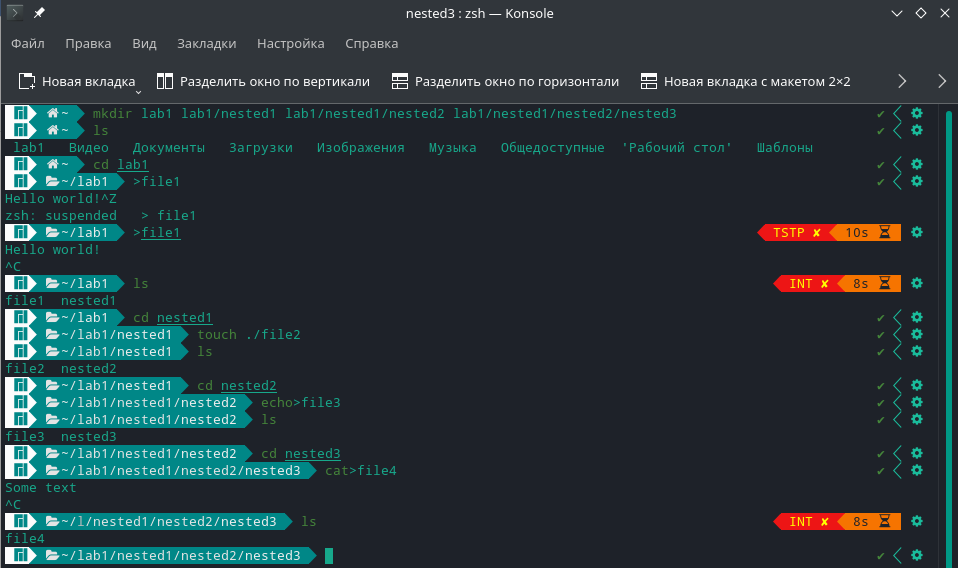
rm – удаляет (ссылки на) файл(ы). Опция -d позволяет удалить пустой каталог



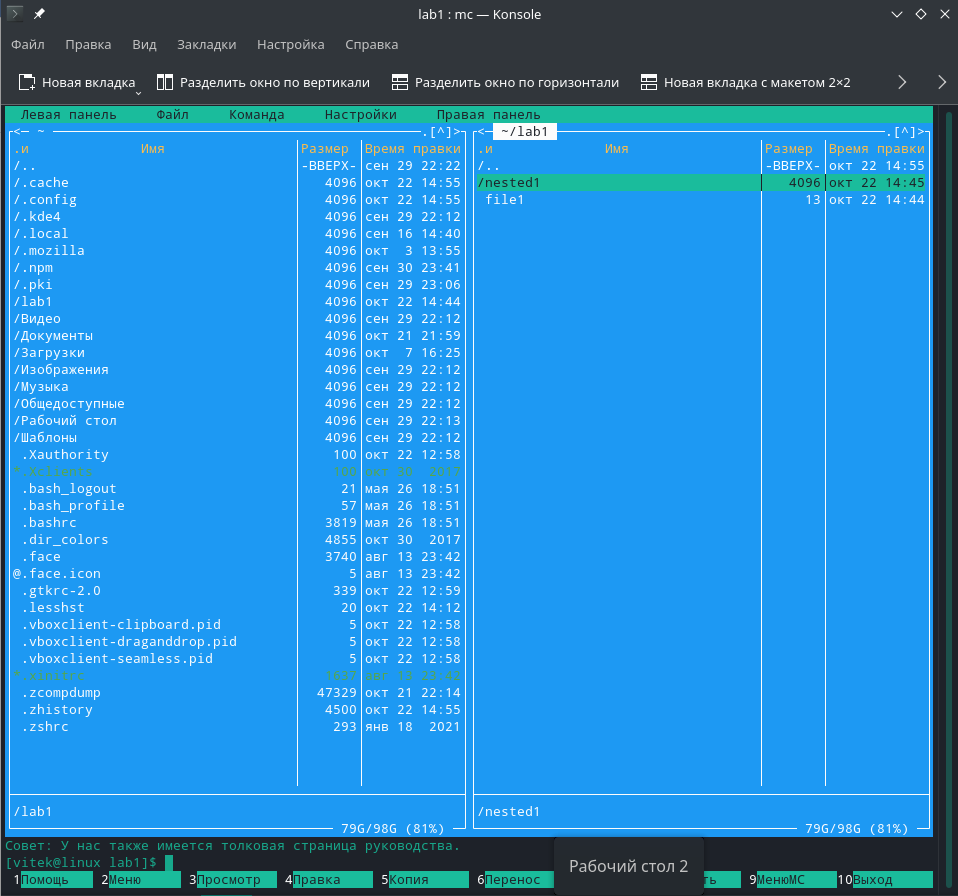
chmod – смена режима доступа к файлам. Например, чтобы сделать файл исполняемым нужно выполнить chmod +x <имя файла>

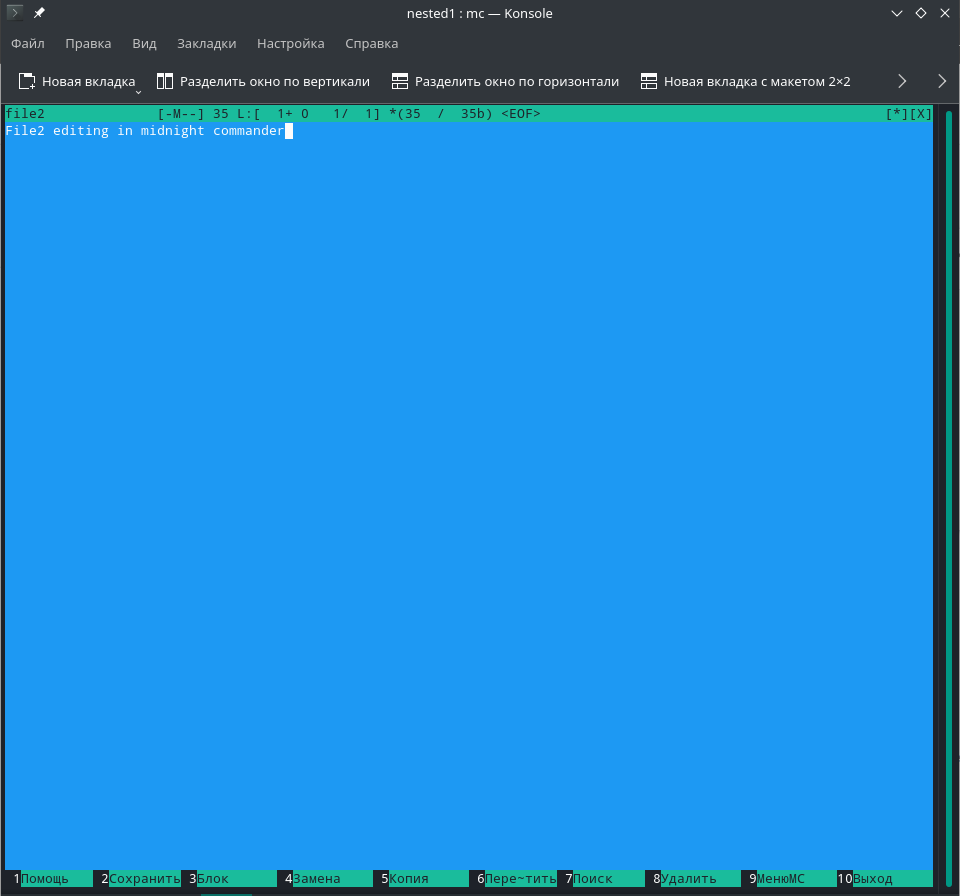


1. Создайте дерево каталогов глубиной вложения до трех уровней, а в самих каталогах создайте текстовые файлы. Примените различные способы создания новых файлов ( >file , touch ./file , echo >file , cat > file)

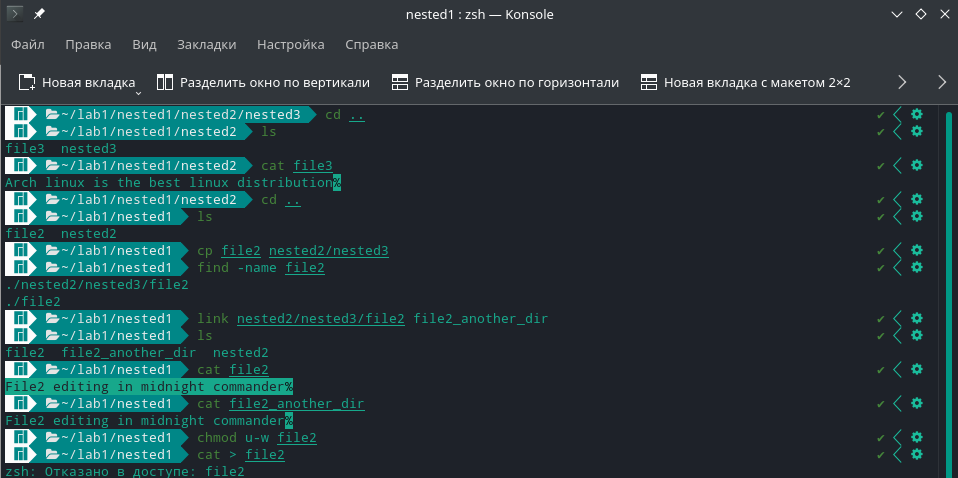


1. Запустите с терминала Midnight Commander вводом команды mc и ознакомьтесь с его основными возможностями по работе с файловой системой. Наполните созданные на предыдущем шаге файлы каким-либо содержанием. Для этого можно использовать любой редактор, от vi, встроенного в ОС, до графического редактора gedit, вызываемого из графической оболочки ОС.





1. Выполните на терминале вторую серию команд cat, cp, find, link, chmod, рассмотренных в лекциях. Для манипуляций с помощью этих команд используйте текстовые файлы, созданные и наполненные на предыдущем шаге



cat file3 - Выводим содержимое файла file3

cd .. – подняться в каталог выше

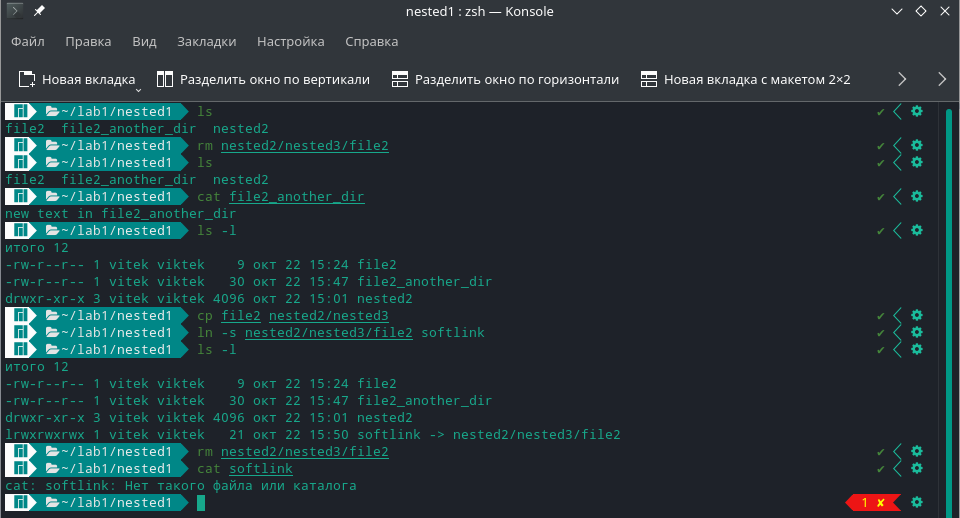
cp file2 nested2/nested3 – копируем file2 в папку lab1/nested1/nested2/nested3

find -name file2 – поиск file2

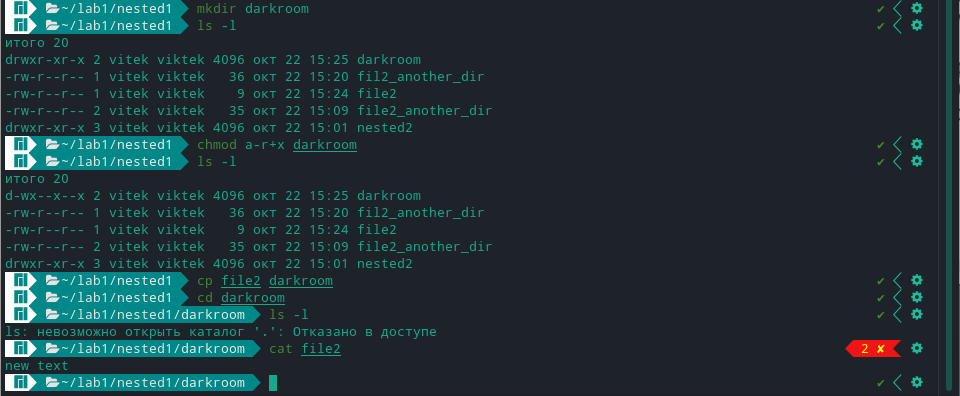
link nested2/nested3/file2 file2\_another\_dir - создаём ссылку на file2 в каталоге lab1/nested1/nested2/nested3 под названием file2\_another\_dir

chmod u-w file2 – для владельца файла убираем разрешение на запись в file2

Жёсткие и мягкие (символьные) ссылки:



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



Создаём каталог darkroom

Получаем его атрибуты

Запрещаем всем пользователям читать, но разрешаем заходить в darkroom

Получаем атрибуты и убеждаемся в том, что предыдущая команда успешно применена

Копируем file2 в darkroom

Переходим в darkroom

Пытаемся прочитать каталог и получить атрибуты для папок и файлов внутри

Получаем отказ

Но вывести на экран содержимое file2 можем (т.к. мы точно знаем, что файл там есть)

**Вывод**

В ходе выполнения работы были получены начальные навыки работы с терминалом системы Linux, был проведен обзор базовых команд ОС, таких как: создание директорий и файлов, открытие и редактирование файлов, создание жестких и мягких ссылок, изменение режима доступа к файлам и директориям, ознакомление с основными возможностями Midnight Commander по работе с файловой системой. В терминале были вызваны основные команды в соответствии с заданием на лабораторную работу (см. «Последовательность выполнения работы»).

# **Лабораторная работа №2 «Работа с процессами в OS Linux»**

**1. Цели работы**

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

**2. Задачи**

1. Войти в систему и скопировать в свой HOME-каталог с разделяемого ресурса набор исходных файлов для второй лабораторной работы.

2. Скомпилировать и выполнить примеры программ forkdemo.cpp, tinymenu.cpp, tinyexit.cpp, procgroup.cpp, wait\_parent.cpp.

3. Модифицировать программу forkdemo.cpp так, чтобы ввод/вывод на терминал отсутствовал, а при переходе по циклу была временная задержка. Запустить модифицированную программу.

4. Исследовать, что произойдет, если процесс-потомок сменит текущий каталог: проверить, будет ли изменен текущий каталог для родителя. Создать программу, подтверждающую ответ и привести её в отчете.

5. Проиллюстрировать как процесс-родитель и процесс потомок разделяют один и тот же дескриптор и смещение текстового файла.

**3. Ход выполнения работы**

1. Войду в систему и скопирую в свой HOME-каталог с разделяемого ресурса набор исходных файлов для второй лабораторной работы. Содержимое файлов приведено в листингах 2.1 – 2.6, соответственно.

|  |
| --- |
| Листинг 2.1 Содержимое файла forkdemo.cpp |
| #include<stdio.h>  #include<sys/types.h>  #include<unistd.h>  main()  {  int i;  if (fork()) { /\* I must be the parent \*/  for(i=0; i<1000 ; i++)  printf("\t\tPARENT %d\n", i);  }  else { /\* I must be the child \*/  for(i=0; i<1000 ; i++)  printf("CHILD %d\n",i);  }  } |

|  |
| --- |
| Листинг 2.2 Содержимое файла tinymenu.cpp |
| /\* The program tinymenu.cpp \*/  #include<stdio.h>  #include<unistd.h>  main()  {  /\* Hard-wired list of commands \*/  static char \*cmd[]={(char \*)"who",(char \*)"ls",(char \*)"date"};  int i;  /\* Prompt for read a command number \*/  printf("0=who, 1=ls, 2=date:");  scanf("%d",&i);  /\* Execute the selected command \*/  execlp(cmd[i], cmd[i], (char \*)0);  printf("Command not found\n");  /\* exec failed \*/  } |

|  |
| --- |
| Листинг 2.3 Содержимое файла tinyexit.cpp |
| /\* The program tinyexit.cpp \*/  #include<stdio.h>  #include<unistd.h>  #include<stdlib.h>  #include<sys/types.h>  #include<sys/wait.h>  main()  {  /\* Hard-wired list of commands \*/  static char \*cmmd[3];  cmmd[0]=(char \*)"who";  cmmd[1]=(char \*)"ls";  cmmd[2]=(char \*)"date";  int i;  while(1){  /\* Prompt for read a command number \*/  printf("0=who, 1=ls, 2=date:");  scanf("%d",&i);  /\* If selection is invalid, parent terminates \*/  if(i<0 || i>2)  exit(2);  if (fork()==0){ /\* Child \*/  /\* The child executes the selected command \*/  execlp(cmmd[i], cmmd[i], NULL);  printf("Command not found\n");  /\* exec failed \*/  exit(1);  }  else  { /\* The parent waits for the child to finish \*/  wait(0);  }  }  } |

|  |
| --- |
| Листинг 2.4 Содержимое файла procgroup.cpp |
| /\*  \* Displaying process group ID information  \*/  #include <stdio.h>  #include<sys/types.h>  #include<unistd.h>  main(){  int i;  printf("\n\nInitial process \t PID %6d \t PPID %6d \t GID %6d\n\n",getpid(), getppid(), getpgid(0));  for (i=0; i<3 ; ++i)  if (fork() == 0) /\* Generate some processes \*/  printf("New process \t\t PID %6d \t PPID %6d \t GID %6d\n", getpid(), getppid(), getpgid(0));  } |

|  |
| --- |
| Листинг 2.5 Содержимое файла wait\_child.cpp |
| /\* The program wait\_child.cpp \*/  #include<stdio.h>  #include<unistd.h>  #include<stdlib.h>  #include<sys/types.h>  #include<signal.h>  main(int argc, char \*argv[])  {  pid\_t pid;  int ret\_value;  pid = getpid();  ret\_value = (int) (pid % 256);  srand((unsigned) pid);  sleep(rand() %5);  if(atoi(\*(argv+1)) % 2){  /\* assuming argv[1] exists! \*/  printf("Child %d is terminating with signal 0009\n", pid);  kill(pid, 9); /\* hara-kiri \*/  }  else{  printf("Child %d is terminating with exit (%04X)\n", pid, ret\_value);  exit(ret\_value);  }  } |

|  |
| --- |
| Листинг 2.6 Содержимое файла wait\_parent.cpp |
| /\* The program wait\_parent.cpp \*/  /\* A parent process waits for a child to finish \*/  #include<stdio.h>  #include<unistd.h>  #include<stdlib.h>  #include<sys/types.h>  #include<sys/wait.h>  main()  {  pid\_t pid, w;  int i, status;  char value[3]; /\*place to store index as string \*/  for(i=0; i<3; ++i)  { /\* Gener. 3 child processes \*/  if ((pid=fork())==0){  sprintf(value, "%d", i);  execl("wait\_child", "wait\_child", value, (char \*)0);  }  else /\* assuming no falures here ...\*/  printf("Forked child %d\n", pid);  }  /\* Wait for the children \*/  while((w=wait(&status)) && w!=-1){  if(w!=-1)  printf("Wait on PID: %d returns status of: %04X\n", w, status);  }  exit(0);  } |

2. Скомпилирую и выполню примеры программ. Начну с forkdemo.cpp с помощью опции g++. Полученный результат программы forkdemo.cpp записывается в файл a.out. Выведу его в терминал. Вывод файла a.out приведен на рисунке 2.1-2.

|  |  |
| --- | --- |
| Рис. 2.1 – Начало a.out | Рис. 2.2 – Фрагмент 1 a.out |
| Рис. 2.3 – Фрагмент 2 a.out | Рис. 2.4 – Фрагмент 3 a.out |
| Рис. 2.5 – Фрагмент 4 a.out | Рис. 2.5 – Конец файла a.out |

Как видно из полученных данных присутствует конкурентность и асинхронность процессов, так как во фрагментах 2, 3, 4, приведенных на рисунках 2.2, 2.3, 2.4 соответственно, наблюдается переключение вывода от процесса к процессу. Важно уточнить, хоть и известен код программы, но невозможно рассчитать то, что будет подано на вывод в определенный момент времени, так как переключение зависит от буферизации, от характеристик компьютера и от кванта времени, который выделяется планировщиком определенному процессу. В коде программы производится вызов fork(), который порождает новый процесс (процесс-потомок), который почти идентичен порождающему процессу-родителю, данный процесс потомок наследует множество параметров, например, сегменты кода, данных и стека программы, таблицу файлов, рабочий и корневой каталоги, реальный и эффективный номер пользователя и номер группы, и другие. Важно отметить, что процесс потомок не наследует от родителя идентификатор процесса (PID, PPID), израсходованное время ЦП (оно обнуляется), сигналов процесса родителя, требующих ответа, блокированных файлов. После вызова возникают два полностью идентичных процесса, весь код после вызова метода fork(), выполняется дважды. В коде есть два цикла с условием for(fork()) {}, так как fork() может возвращать три параметра: -1 – при вызове fork() возникла ошибка, 0 – это код процесса-потомка, 1 – код процесса-родителя.

Продолжу процесс компиляции файл tinymenu.cpp. Выполню файл a.out и при этом выведу результат в терминал. Полученные данные в терминале приведены на рисунке 2.6. В программе tinymenu используется вызов системного метода exec(), который позволяет передавать параметры командной строки, но в то же время переменные окружения будут наследоваться, их может быть несколько, по соглашению первый параметр должен указывать на имя, ассоциируемое с файлом, который необходимо исполнить. С помощью вызова exelp() в примере запускаются команды интерпретатора. Содержимое терминала при введении параметров равных 0, 1, 2 представлено на рисунке 2.7.

|  |
| --- |
| Рис. 2.6 – Содержимое файла a.out при компиляции tinymenu.cpp |
| Рис. 2.7 – Вывод программы при введении параметров |

Далее произведу компиляцию файла tinyexit.cpp. Так как tinyexit содержит системный процесс wait, то родительский процесс переводится в состояние ожидания, после, если выполняется вызов метода exit(), то происходит выход по событию завершению родительского процесса. Вывод файла a.out приведен на рисунке 2.9. Содержимое терминала представлено на рисунке 2.10 при введении параметров 0, 1, 2, 3.

|  |
| --- |
| Рис. 2.9 – Вывод файла a.out после компиляции tinyexit.cpp |
| Рис. 2.10 – Вывод в терминал при введении параметров |

Далее скомпилирую и выполню программу procgroup.cpp. Данная программа отображает информацию о процессах. Вывод терминала представлен на рисунке 2.11. Вызов метода getpgid() аналогичен методу getpgid(0) – получает группу процессов.

|  |
| --- |
| Рис. 2.11 – Вывод работы программы procgroup.cpp |

Далее скомпилирую и выполню программу wait\_parent.cpp. В данной программе запускается три потомка из родительского процесса и отслеживается завершения каждого из них с помощью вызова системного метода wait(). Вывод в терминал представлен на рисунке 2.12. Как видно также наблюдается информация о том, какой код завершения процесса был передан. В данном примере wait\_child – процесс потомок, которая запускается из wait\_parent при помощи вызова метода exelp c разными параметрами. В процессах wait\_child формируется код завершения. Важно уточнить, что при завершении по сигналу, номер сигнала передается wait\_parent в младшем байте статуса завершения.

|  |
| --- |
| Рис. 2.12 – Вывод в терминал программы wait\_parent.cpp |

Далее скомпилирую и выполню программу wait\_child.cpp. Содержимое вывода wait\_parent представлено на рисунке 2.13. Как видно, что при завершении по сигналу, номер сигнала передается в младшем байте статуса разряда.

|  |
| --- |
| Рис. 2.13 – Вывод выполнения программы wait\_child.cpp и wait\_parent.cpp |

3. Модифицирую программу forkdemo().cpp так, чтобы не было ввода и вывода в терминал, а при проходе по циклу установлю временную задержку sleep(10). Запущу данную программу в фоновом режиме (background), введя при запуске символ & после пробела и зафиксировав значение PID, назначенное системой фоновому процессу при запуске. Выполню в терминале команды ps, top, uptime, pstree. Код модифицированной программы приведен в листинге 2.7.

|  |
| --- |
| Листинг 2.7 – Код программы forkdemo\_m.cpp |
| #include<stdio.h>  #include<sys/types.h>  #include<unistd.h>  main()  {  pid\_t PID;  PID = fork();  int i;  if (PID) { /\* I must be the parent \*/  for(i=0; i<1000 ; i++)  sleep(10);  }  else { /\* I must be the child \*/  for(i=0; i<1000 ; i++)  sleep(10);  }  } |

Запущу программу в фоновом режиме с помощью инструкции, описанной выше. Выполнение программы и результаты представлены на рисунке 2.14. Как можно убедиться программа ничего не выводит и не требует ввести что-либо.

|  |
| --- |
| Рис. 2.14 – Содержимое терминала при выполнении forkdemo\_m.cpp |

Выполню команды ps и ps aux, которые выводят список процессов в терминал. Результаты представлены на рисунках 2.15 и 2.16 соответственно.

|  |
| --- |
| Рис. 2.15 – Выполнение команды ps |
| Рис. 2.16 – Выполнение команды ps aux |

Выполню команду top, которая позволяет наблюдать за процессами в режиме реального времени. Результат выполнения приведен на рисунке 2.17.

|  |
| --- |
| Рис. 2.17 – Выполнение команды top |

Выполню команду uptime, которая позволяет посмотреть на информацию о загрузки системы. Результат выполнения приведен на рисунке 2.18.

|  |
| --- |
| Рис. 2.18 – Выполнение команды uptime |

Выполню команду pstree, чтобы посмотреть все запущенные процессы. Результат выполнения представлен на рисунке 2.19.

|  |
| --- |
| Рис. 2.19 – Выполнение команды pstree |

Сниму собственный процесс с помощью команды kill и параметров процесса, и с помощью команды w можно посмотреть список активных процессов для всех пользователей – можно убедиться, что процесс завершен. Результаты представлены на рисунках 2.20 и 2.21.

|  |
| --- |
| Рис. 2.20 – Выполнение команды kill |
| Рис. 2.21– Выполнение команды w |

4. Исследую ситуацию, когда процесс-потомок меняет текущий каталог. Создам программу, моделирующую данную ситуацию. Код программы представлен в листинге 2.8.

|  |
| --- |
| Листинг 2.8 – Код программы DIR\_change.cpp |
| #include<stdio.h>  #include<sys/types.h>  #include<unistd.h>  main()  {  int i;  if (fork()) { /\*I must be the parent \*/  printf("\n %s \n", get\_current\_dir\_name());  for(i=0; i<20 ; i++)  printf("\t\tPARENT %d\n", i);  sleep(10);  }  else { /\*I must be the child \*/  printf("\n %d \n", chdir("home/alex/OS"));  printf("\n %s \n", get\_current\_dir\_name());  for(i=0; i<20 ; i++)  printf("CHILD %d\n",i);  sleep(10);  }  return 0;  } |

Скомпилирую и запущу программу DIR\_change.cpp. Результаты выполнения представлены на рисунке 2.22. Как видно из полученных результатов программы при корректном перемещении процесса-потомка каталог процесса родителя не изменился, что и было описано выше при описании команды fork(). Поэтому данная операция не влияет на потоки процессов.

|  |
| --- |
| Рис. 2.22 – Результаты выполнения программы DIR\_change.cpp |

5. Необходимо проиллюстрировать, как процесс-родитель и процесс-потомок разделяют один и тот же дескриптор и смещение текстового файла. Чтобы исследовать данную ситуацию, необходимо составить программу, где процесс-родитель открывает некий текстовый файл и запускает потомка, в то же время процесс-потомок читает из данного файла фрагмент данных и выводит их в консоль. По завершению потомка родитель должен читать из того же файла и выводить результат в консоль. Чтобы синхронизировать доступ родителя и потомка к файлу буду использовать метод sleep(). Код программы приведен на листинге 2.9. Содержимое input.txt приведено на рисунке 2.23. Результат выполнения приведен на рисунке 2.24. Как можно наблюдать, программа работает корректно, а это означает, что при определении процесса-потомка ему наследуется таблица файлов, в которой находятся состояния флагов дескрипторов файла, указывающие, читается ли файл или пишется. Кроме того, в таблице файлов содержится текущая позиция указателя записи-чтения.

|  |
| --- |
| Листинг 2.23 – Код программы test\_input.cpp |
| #include <stdio.h>  #include <stdlib.h>  #include <sys/types.h>  #include <sys/wait.h>  #include <unistd.h>  int main() {  FILE \*ptr;  size\_t length = 0;  ssize\_t up;  char\* buf = (char\*)malloc (100\*sizeof(char));  if ((ptr = fopen("input.txt", "r")) == NULL) {  printf("File error occured\n");  exit(1);  }  if (fork() == 0){  if((up = getline(&buf, &length, ptr)) != -1) printf("Child: %s", buf);  exit(0);  } else {  wait(NULL);  if((up = getline(&buf, &length, ptr)) != -1) printf("Parent: %s", buf);  fclose(ptr);  }  free(buf);  return 0;  } |

|  |
| --- |
| Рис. 2.23 – Содержимое файла input.txt |
| Рис. 2.24 – Выполнение программы test\_input.cpp |

**4. Вывод**

В ходе выполнения работы были изучены и испытаны на практике основные методы работы с процессами. Осуществлен запуск, порождение процессов-потомков, вызов методов ожидания, и прерывания процесса. Были изучены основные их свойства, а именно свойства директорий текущих процессов, работа с идентификаторами процессов и файлов

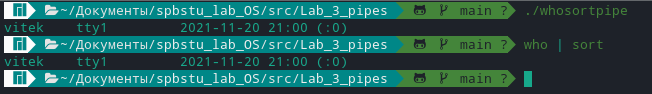
# **Лабораторная работа №3 «Программные каналы»**

**Цель работы**

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

**Последовательность выполнения работы:**

1. Войдите в систему и скопируйте с разделяемого ресурса в свой HOMEкаталог набор исходных файлов для третьего занятия.
2. Скомпилируйте и выполните программу whosortpipe.cpp. Сопоставьте результат выполнения программы с выполнением этих же двух команд из shell в конвейерном режиме ( | ). Не забывайте приводить в отчете анализ результатов работы этой программы (как и всех последующих) с соответствующими скриншотами.

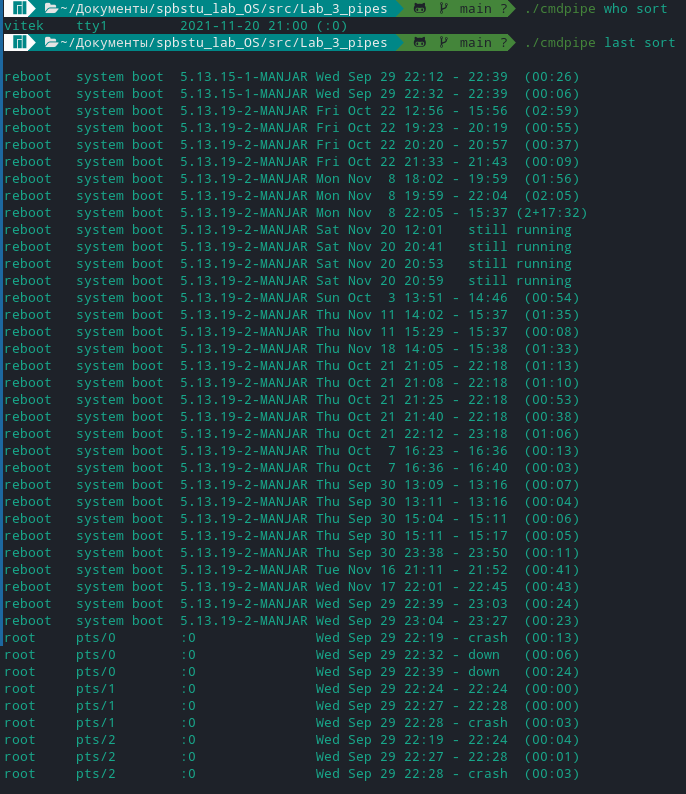


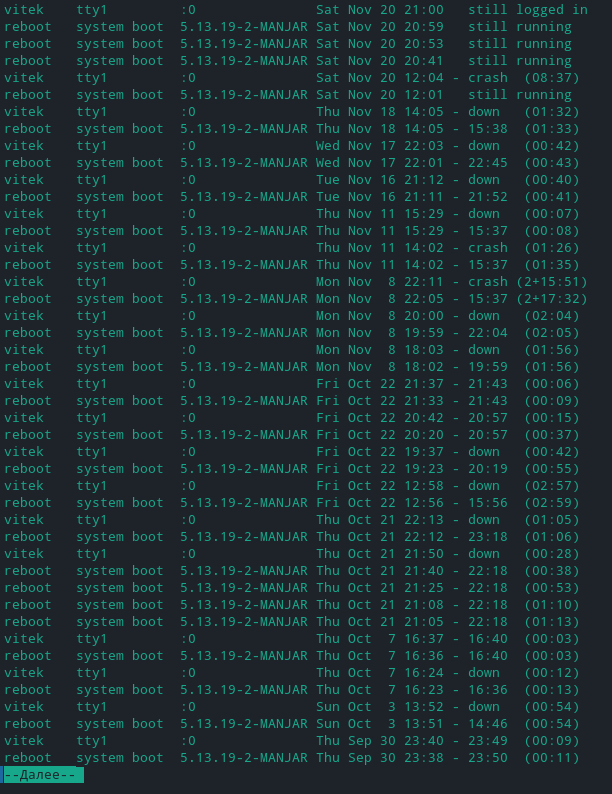
*sort* - простая и очень полезная команда, которая меняет порядок строк в текстовом файле, то есть осуществляет их сортировку по алфавиту или в соответствии с числовыми значениями.

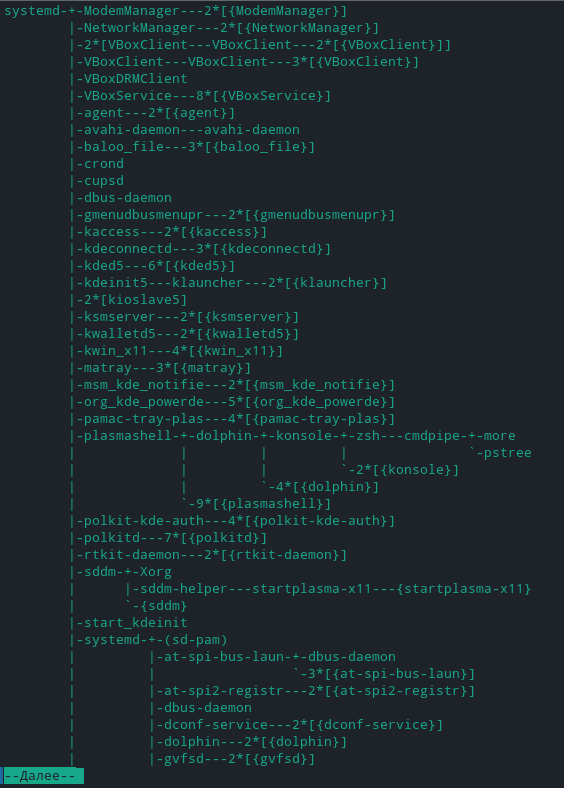
**$ sort опции файл или $ команда | sort опции**

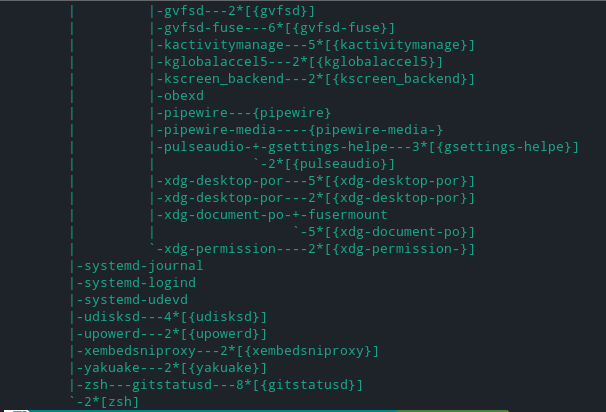
*who -* вывести информацию о пользователях, вошедших в систему.

1. 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 команд из shell в конвейерном режиме. Можно ли с помощью вызова popen() 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.









*more -* предназначена для постраничного просмотра файлов в терминале Linux. Своим названием она обязана надписи more (в русскоязычном варианте — дальше), появляющейся внизу каждой страницы.

**$ more опции файл**

*last* - утилита командной строки, которая отображает информацию о последних сеансах входа пользователей системы. Это очень полезно, когда вам нужно отслеживать активность пользователя или расследовать возможное нарушение безопасности.

**$ last опции пользователь <TTY>…**

*less -* позволяет перематывать текст не только вперёд, но и назад, осуществлять поиск в обоих направлениях, переходить сразу в конец или в начало файла.

**$ less опции файл**

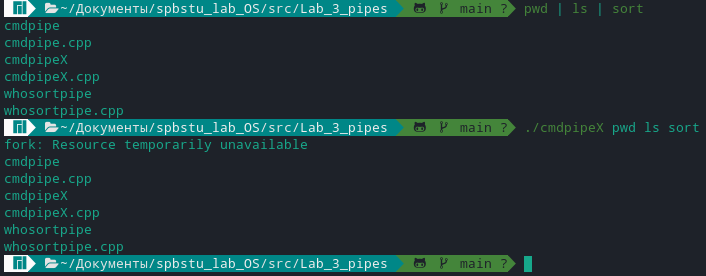
С помощью вызова popen() нельзя создать программу, организующую конвейер из трех команд shell, т.к. метод popen() с аргументом w может только подавать данные на вход указанной команды, но не может считывать ее вывод (результат работы команды автоматически подается на стандартный вывод).

1. Напишите программу (например, на основе вызовов pipe()), 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:

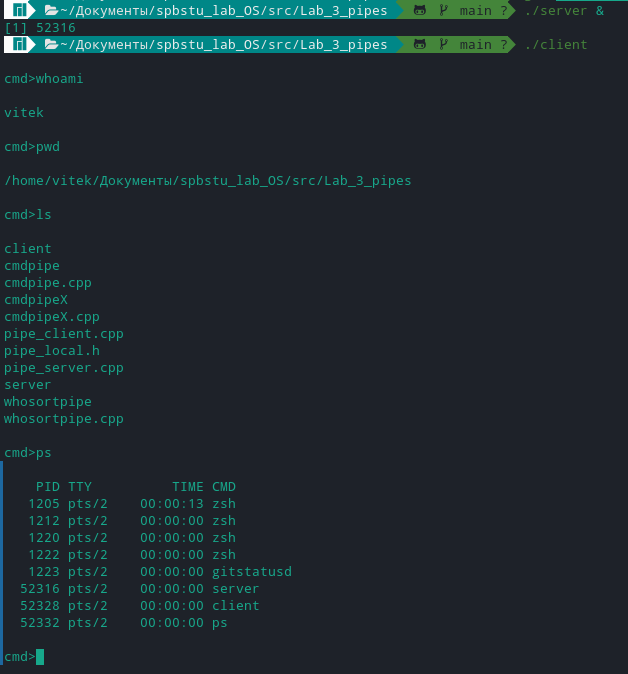
$ last | sort | more.



Программа:

|  |
| --- |
| /\*  \*\* pipex.c - multipipes support  \*/  #include <stdlib.h>  #include <unistd.h>  #include <stdio.h>  #include <sys/wait.h>  /\*  \* loop over commands by sharing  \* pipes.  \*/  static void  pipeline(char \*\*\*cmd)  {  int fd[2];  pid\_t pid;  int fdd = 0; /\* Backup \*/  while (\*cmd != NULL) {  pipe(fd);  if ((pid = fork()) == -1) {  perror("fork");  exit(1);  }  else if (pid == 0) {  dup2(fdd, 0);  if (\*(cmd + 1) != NULL) {  dup2(fd[1], 1);  }  close(fd[0]);  execvp((\*cmd)[0], \*cmd);  exit(1);  }  else {  wait(NULL); /\* Collect childs \*/  close(fd[1]);  fdd = fd[0];  cmd++;  }  }  }  /\*  \* Compute multi-pipeline based  \* on a command list.  \*/  int  main(int argc, char \*argv[])  {  char \*temp[argc][2];  for (int i = 0; i < argc; i++)  {  temp[i][0] = argv[i];  temp[i][1] = NULL;  }  char \*ls[] = {"ls", NULL};  char \*rev[] = {"rev", NULL};  char \*nl[] = {"nl", NULL};  char \*cat[] = {"cat", "-e", NULL};  char \*\*cmd[argc + 1];  for (int i = 0; i < argc; i++)  {  cmd[i] = temp[i];  }  cmd[argc] = NULL;  pipeline(cmd);  return (0);  } |

1. 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 результаты функционирования данной системы и 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.



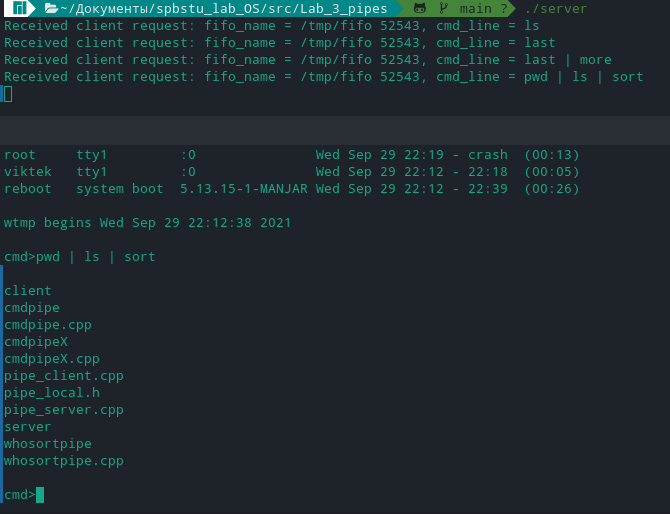
Так как сервер выполняет команды последовательно, то общая скорость работы программы зависит от скорости выполнения каждого из процессов. Такая пропорциональность времени отклика сервера чревата его чрезмерной нагрузкой при трудоемких запросах.

Также, если не запустить сервер, то клиент упадёт.

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 (выполнения переданной от клиента команды и переправки клиенту результата).

Измененная программа pipe\_server , при получении сообщения , операция выполняется и выводится в процессе потомке.

|  |
| --- |
| /\* The server program pipe\_server.cpp \*/  #include"pipe\_local.h"  #include <sys/wait.h>  int publicfifo;  void sigint\_handler(int sig) {  if (sig == SIGINT) {  close(publicfifo);  remove(PUBLIC);  exit(0);  }  }  int main(void)  {  const int max\_forks = 10;  int n, done, dummyfifo, privatefifo;  static char buffer[PIPE\_BUF];  FILE \*fin;  struct message msg;  /\* Generate the public FIFO \*/  mknod(PUBLIC, S\_IFIFO | 0666, 0);  struct sigaction sa;  sa.sa\_handler = sigint\_handler;  sa.sa\_flags = 0;  sigemptyset(&sa.sa\_mask);  if (sigaction(SIGINT, &sa, NULL) == -1) {  perror("sigaction");  exit(1);  }  /\* OPEN the public FIFO for reading and writing \*/  if ((publicfifo=open(PUBLIC, O\_RDONLY))==-1 ||  (dummyfifo=open(PUBLIC, O\_WRONLY | O\_NDELAY))==-1){  perror(PUBLIC);  exit(1);  }  /\* Message can be read from the PUBLIC pipe \*/  int fork\_count = 0;  int pid = 0;  while(read(publicfifo, (char \*) &msg, sizeof(msg))>0){  if (fork\_count < max\_forks) {  pid = fork();  if (pid == 0) {  fork\_count++;  n = done = 0; /\* Clear counters | flags \*/  printf("Received client request: fifo\_name = %s, cmd\_line = %s", msg.fifo\_name, msg.cmd\_line);  do{ /\* Try OPEN of private FIFO \*/  if ((privatefifo=open(msg.fifo\_name, O\_WRONLY | O\_NDELAY))==-1)  sleep(3); /\* Sleep a while \*/  else{ /\* OPEN succesful \*/  fin = popen(msg.cmd\_line, "r"); /\* Execute the cmd \*/  write(privatefifo, "\n", 1); /\* Keep output pretty \*/  while((n=read(fileno(fin), buffer, PIPE\_BUF))>0){  write(privatefifo, buffer, n); /\*to private FIFO \*/  memset(buffer, 0x0, PIPE\_BUF); /\* Clear it out \*/  }  pclose(fin);  close(privatefifo);  done = 1; /\* Record succes \*/  }  }while(++n<5 && !done);  if(!done) /\* Indicate failure \*/  write(fileno(stderr), "\nNOTE: SERVER \*\* NEVER \*\* accessed private FIFO\n", 48);  } else {  wait(0);  fork\_count--;  }  }  }  return 0;  } |



**Вывод**

В ходе лабораторной работы были исследованы способы организации каналов. Изучено понятие конвейер и его свойства. Получены навыки запуска команд по конвейеру и была написана программа для запуска конвейером нескольких команд. Также разобран пример клиент-серверного взаимодействия.

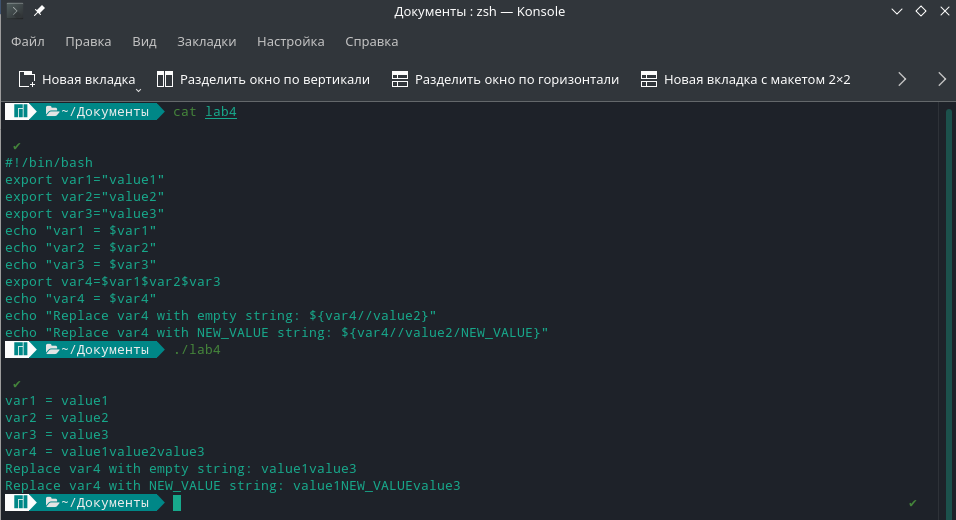
# **Лабораторная работа №4 «Командные файлы. Переменные окружения»**

**Цель работы**

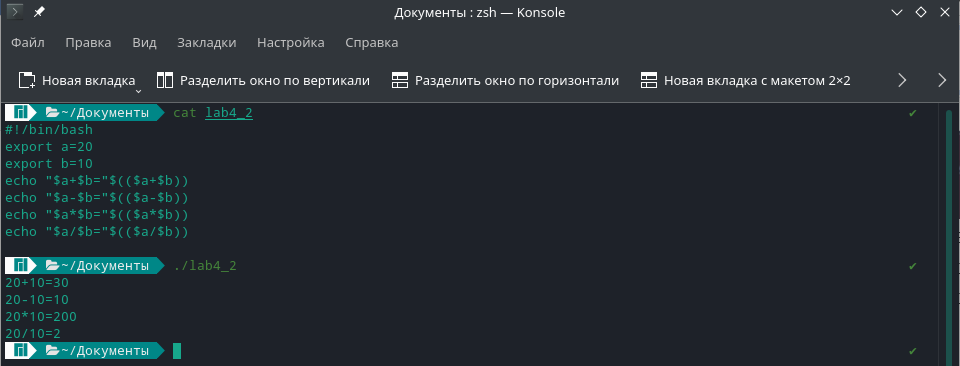
Знакомство с важным атрибутом любой операционной системы – переменными среды (или переменными окружения) и с возможностями их использования в Linux. Освоение языка для составления командных сценариев и написание набора полезных для системного администрирования скриптов.

**Последовательность выполнения работы:**

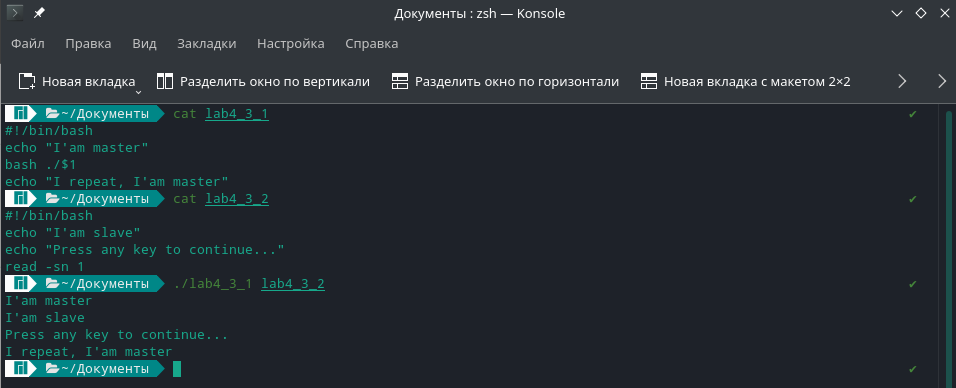
1. Создайте несколько символьных переменных среды (переменных окружения). Составьте командный файл (сценарий bash), выводящий на консоль значения этих переменных. Выполните операцию конкатенации (склеивания) значений переменных и выведите полученный результат на консоль. Выделите из конкатенированной переменной среды подстроку и выведите ее на консоль. Замените выделенную подстроку на какое-либо другое значение и выведите измененное значение переменной среды на консоль.



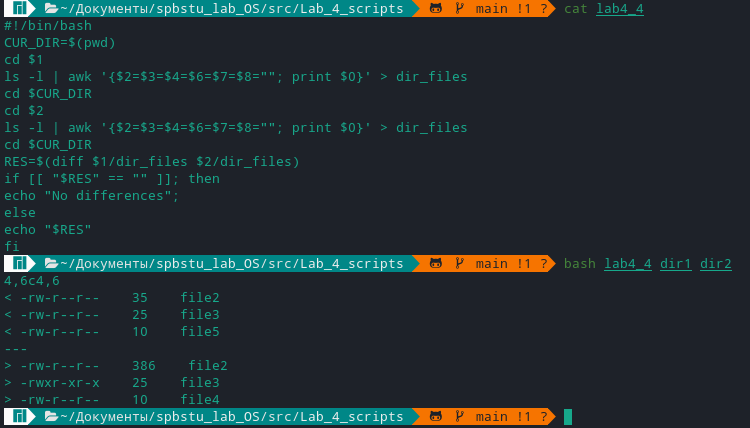
1. Создайте несколько переменных среды в интерпретации, как числовые переменные. В новом командном файле выполните с этими числовыми переменными все допустимые арифметические операции, выводя на консоль результаты операций и соответствующие комментарии.



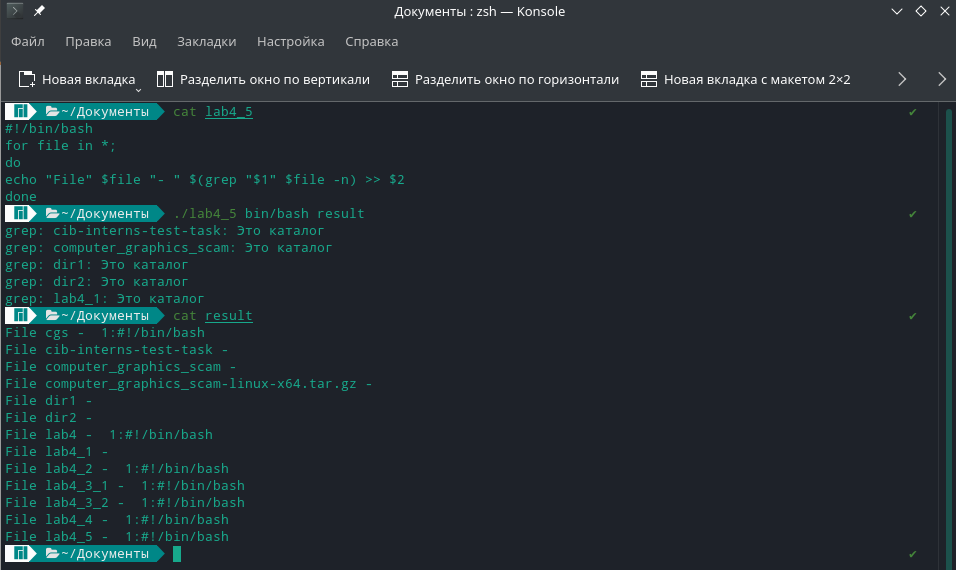
1. Создайте командный файл (основной), выдающий при старте сообщение и затем вызывающий другой командный файл (его имя задается при старте основного файла в качестве параметра командной строки), который выдает свое сообщение и приостанавливается до нажатия любой клавиши. При возврате управления в вызывающий (основной) файл из него должно выдаваться еще одно сообщение, подтверждающее возврат.



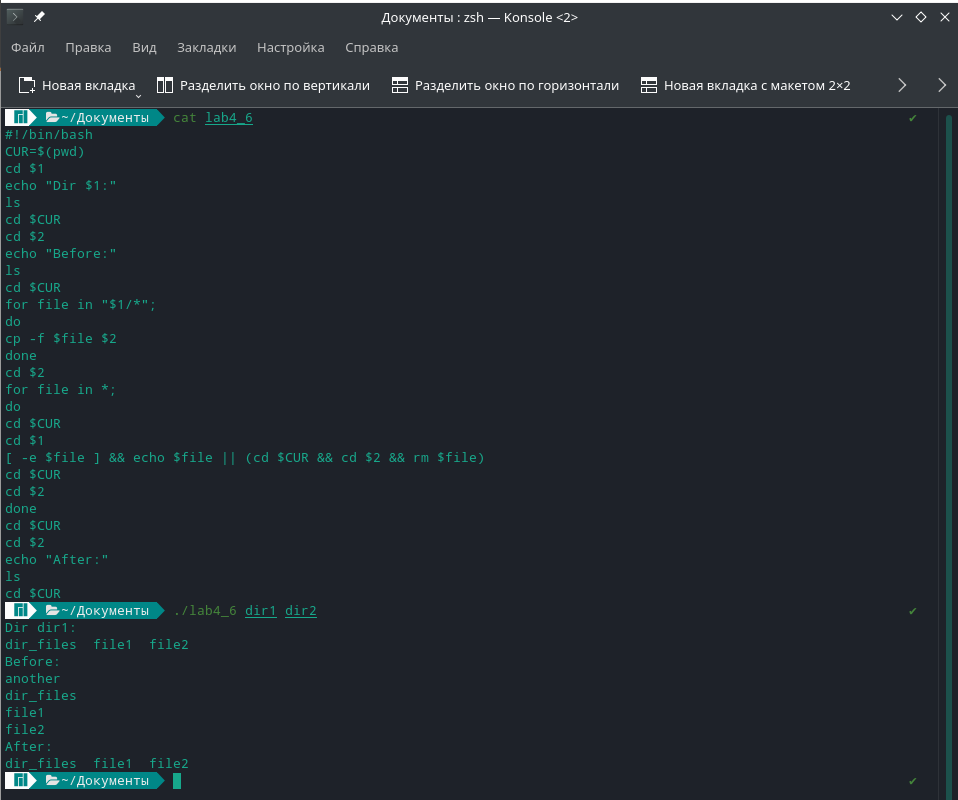
1. Составьте командный файл, выводящий на экран различия содержимого двух каталогов, имена которых передаются в качестве параметров. Отличия искать в именах файлов, их размерах и атрибутах.



1. Разработайте командный файл сценария для поиска текстовых файлов, содержащих заданную последовательность символов. Эта последовательность передается при запуске в качестве первого параметра командной строки. В качестве второго параметра передается имя файла результатов, который должен быть создан в сценарии для записи в него имен найденных текстовых файлов и номеров их строк, в которых содержится заданная последовательность символов.



1. Создайте командный файл, который синхронизирует содержимое заданного каталога с эталонным. После запуска и отработки командного файла в заданном каталоге должен оказаться тот же набор файлов, что и в эталонном (если файла нет – он копируется из эталонного каталога, если найдется файл, которого нет в эталонном, – удаляется). Если файл с некоторым именем есть и в заданном, и в эталонном каталогах, то он перезаписывается только в том случае, если в эталонном имеется более новая версия файла. Имена обоих каталогов должны при запуске передаваться командному файлу параметрами командной строки.



Все файлы из директории 3 копируются в 4. После этого просматриваются файлы 4 директории и удаляются те, которых нет в 3.

Программа выводит в терминал имена файлов, которые не были удалены при синхронизации.

**Вывод**

В ходе лабораторной работы было ознакомление с переменными среды и с возможностями их использования в Linux. Также получено освоение языка для составления командных сценариев и написание набора полезных для системного администрирования скриптов.

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

**1. Цели 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.

**2. Задачи**

1. Создать несколько учетных записей для нескольких пользователей (не задавая им прав администратора) и объедините их в две группы. Заходя в систему под разными аккаунтами, создайте в соответствующих домашних каталогах файлы, варьируя при этом права доступа для пользователя, для группы, для всех. Убедиться, что права доступа разделяются в соответствии с тем, как это задано. Провести операцию слияния файлов с различными правами доступа и проверить, какие при этом получаются права у результирующего файла.

2. Запустить в фоновом режиме командный файл (процесс), выдающий в цикле с некоторой задержкой сообщение на консоль. Запустить другой командный файл (процесс), требующий диалога, в обычном режиме. Убедиться, что вывод двух этих процессов на консоль перемежается. Остановить фоновый процесс сигналом kill. Запустить его снова, организовав предварительно перенаправление его вывода в файл. Убедиться, что теперь вывод двух процессов разделен.

3. Доработать предыдущее задание так, чтобы показать возможность перевода фонового процесса в диалоговый режим для выполнения операции ввода с клавиатуры и затем возврата его обратно в фоновый режим. Продемонстрировать возможность оставления фонового процесса на исполнение после завершения пользовательского сеанса работы в ОС.

4. Разработать командный файл для выполнения архивации каталога через определенные интервалы времени. Запустить командный файл в режиме background. Имя архивируемого каталога, местоположение архива и время архивации передаются при запуске командного файла в виде параметров

**3. Ход выполнения работы**

1. Создам несколько учетных записей для нескольких пользователей (не задавая им прав администратора) и объединю их в две группы. На рисунке 5.1 представлено сколько пользователей имеется на данный момент. Создам два пользователя user1 и user2. Создание приведено на рисунке 5.2. Задам пользователям пароли, операция приведена на рисунке 5.3. Список обновленных пользователей приведен на рисунке 5.3. Также было создано две группы и них добавлены пользователи, результат приведен на рисунке 5.4. Под пользователями теперь можно авторизоваться. Выбор пользователей приведен на рисунке 5.5. Зайду под каждым новым пользователем и создам текстовый файл в домашней директории.

|  |
| --- |
| Рис. 5.1 – Выполнение команды ls -l /home |
| Рис. 5.2 – Создание двух пользователей и задание им пароля |
| Рис. 5.3 – Обновленный список пользователей |
| Рис. 5.4 – Создание двух групп и добавление туда пользователей |
| Рис. 5.5 – Выбор пользователя |

Зайду в директории созданных пользователей и посмотрю все файлы, которые содержит каталог новых пользователей. Результат выполнения приведен на рисунке 5.6.

|  |
| --- |
| Рис. 5.6 Просмотр директорий других пользователей |

Из полученных результатов видно, что у файлов, который были созданы другими пользователями, нет флагов для модификации файла. Записана комбинация -rw-rw-r--, которая говорит, что пользователь и его группа может редактировать и читать файл, а остальные только read only. Проверю, что будет при слиянии файлов с разными правами доступа. Для этого выполню команду cat слияния двух файлов и помещу полученный файл в директорию оригинального пользователя. Результаты приведены на рисунке 5.7. Зайду в каталог пользователя и проверю права доступа, изображение приведено на рисунке 5.8.

|  |
| --- |
| Рис. 5.7 – Слияние двух файлов user1 и user2 |
| Рис. 5.8 – Права доступа к файлу file.txt |

Видно, что редактировать файл может пользователь и его группа, поэтому при слиянии двух файлов, права на редактирование получает пользователь, который выполнил слияние.

2. Создам два командных файла, который приведены в листингах 5.1 и 5.2, которые выводят сообщения в терминал. Запущу первый файл в фоновом режиме. Второй же файл запущу в режиме foreground. Результаты вывода в терминал приведены на рисунке 5.9. По полученным данным видно, что данные перемежаются. Остановлю фоновый процесс сигналом kill c соответствующим параметром. Модифицирую файл, чтобы запись была перенаправлена в определенный файл. Для этого достаточно указать то, куда будут записываться строки: echo “информация”>>text.txt. запуск программ в фоновом режиме приведен на рисунке 5.10. Содержимое текстовых файлов представлено на рисунках 5.11 И 5.12. Из полученных результатов можно сделать вывод, что вывод двух файлов разделен.

|  |
| --- |
| Листинг 5.1 – script1.sh |
| #!/bin/bash  COUNTER=0  echo "its the first script"  while [ $COUNTER -lt 1000 ]  do  echo "in first script"  sleep 10  COUNTER=$(($COUNTER+1))  done |

|  |
| --- |
| Листинг 5.2 – script2.sh |
| #!/bin/bash  COUNTER=0  echo "its the second script"  while [ $COUNTER -lt 1000 ]  do  echo "in second script"  sleep 10  COUNTER=$(($COUNTER+1))  done |

|  |
| --- |
| Рис. 5.9 – Результаты работы двух командных файлов |
| Рис. 5.10 – Запуск в фоновом режиме модифицированных файлов |

|  |
| --- |
| Рис. 5.11 – Содержимое файла text1.txt через некоторое время |
| Рис. 5.12 – Содержимое файла text2.txt через некоторое время |

3. Доработаю предыдущее задание так, чтобы показать возможность перевода фонового процесса в диалоговый режим для выполнения операции ввода с клавиатуры и затем возврата его обратно в фоновый режим. Для этого необходимо, чтобы командный файл запрашивал информацию у пользователя. Для этого пропишу в файле дополнительно echo -n “Введите информацию” и read info, а потом выведу через echo “$info”. Результат выполнения в диалоговом режиме представлен на рисунке 5.13.

|  |
| --- |
| Рис. 5.13 – Результат выполнения script3.sh с введением информации |

Для тестирования запущу процесс в фоновом режиме и переведу его в диалоговый для тестирования ввода информации. Результаты приведены на рисунке 5.14. Сначала запускаю процесс в диалоговом режиме. Проверяю работоспособность, перевожу процесс в фоновый режим, восстанавливаю его и проверяю работоспособность, можно сказать, что процесс можно перевести обратно в диалоговый режим и продолжить вводить информацию с клавиатуры.

|  |
| --- |
| Рис. 5.14 – Тестирование ввода при возвращении в диалоговый режим |

Протестирую ситуацию, когда пользователь завершает сеанс. Для этого запущу процесс, переведу его в фоновый режим, завершу сеанс пользователя, при повторном входе проверю работоспособность процесса. При вызове команды jobs не было отображено каких-либо процессов, а это означает, что при завершении сеанса или закрытии терминала фоновые процессы завершаются. Важно отметить, что данную проблему можно решить с помощью создания процессов, используя tmux, так как сеансы tmux являются постоянными и поэтому даже при закрытии терминала программы продолжают работать. Важно уточнить, что при смене пользователя процесс продолжает свою работу.

5. Необходимо разработать командный файл, который архивирует каталог через определенные интервалы времени. Листинг кода приведен в листинге 5.3. Запущу командный файл в фоновом режиме. Результаты приведены на рисунке 5.15. Наличие архивированного каталога приведено на рисунке 5.16. По полученным результатам видно, что фоновый процесс работает, tar успешно архивирует нужный каталог и выводит сообщение о том, какие файлы были помещены в архив. Пользователь третьим параметром задает период архивации. Важно отметить, что архив либо создается, если в папке назначения не было архива с тем же именем, либо перезаписывает сам себя.

|  |
| --- |
| Листинг 5.3 – Содержимое файла scriptForArchive.sh |
| #!/bin/bash  isWorking=1  echo "start to archive"  name\_of\_catalogue=$1  way\_to\_archive=$2  period=$3  while [ $isWorking ]  do  cd ~  zip -r archive.zip $1  mv archive.zip $2  sleep $3  done |

|  |
| --- |
| Рис. 5.15 – Тестирование архивации фоновым процессом |
| Рис. 5.16 – Архивированный каталог |

**4. Вывод**

В ходе выполнения работы были получены навыки создания пользователей, определения прав к файлам, также были определены права файла, который был получен при слиянии нескольких файлов с разными правами доступа. Также в лабораторной работе были написаны командные файлы, которые помогли разобраться с фоновыми и диалоговыми режимами работы. При этом был создан скрипт, который архивирует каталог с определенной периодичностью и протестирован на реальном примере.

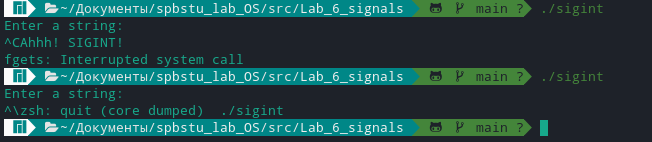
# **Лабораторная работа №6 «Генерация и обработка сигналов»**

**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.

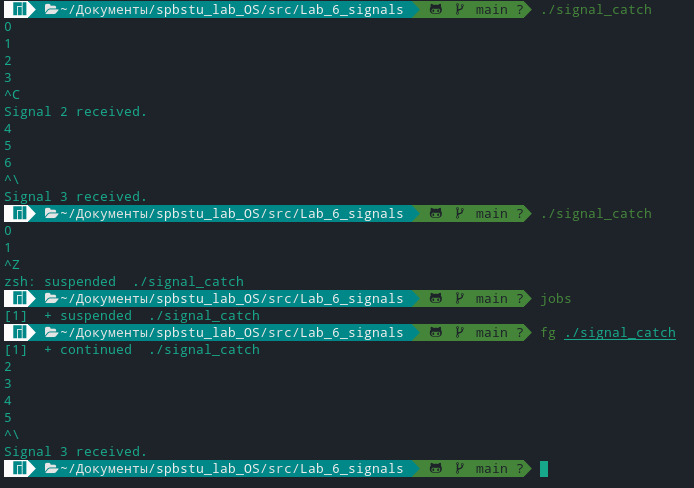
**Последовательность выполнения работы:**

1. Войдите в систему и скопируйте с разделяемого ресурса в свой HOME-каталог набор исходных файлов для шестого занятия.
2. Программа sigint.cpp осуществляет ввод символов со стандартного ввода. Скомпилируйте и запустите программу и отправьте ей сигналы SIGINT (нажатием Ctrl-C) и SIGQUIT (нажатием Ctrl-\). Проанализируйте результаты.



Программа перехватывает сигнал SIGINT, чтобы вывести соответствующее сообщение перед завершением.

1. 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.



SIGINT – завершение процесса

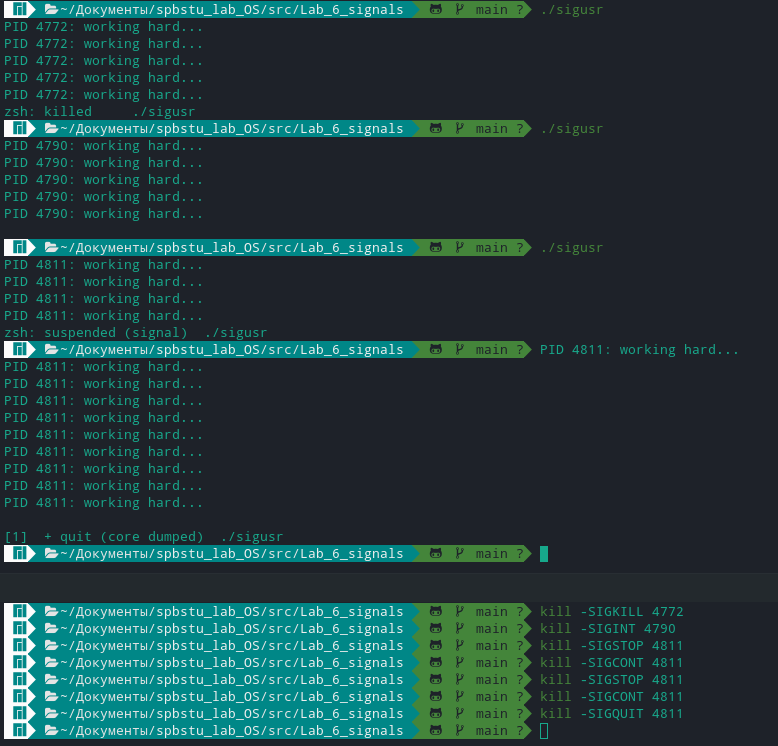
SIGQUIT – остановка процесса + выполнение дампа памяти для процесса (срез данных непосредственно из самой памяти, из ОЗУ)

SIGSTOP – остановка процесса

SIGCONT – запуск остановленного процесса

От прошлой программы эта отличается тем, что при получении SIGINT программа не завершается, то есть этот сигнал игнорируется.

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.



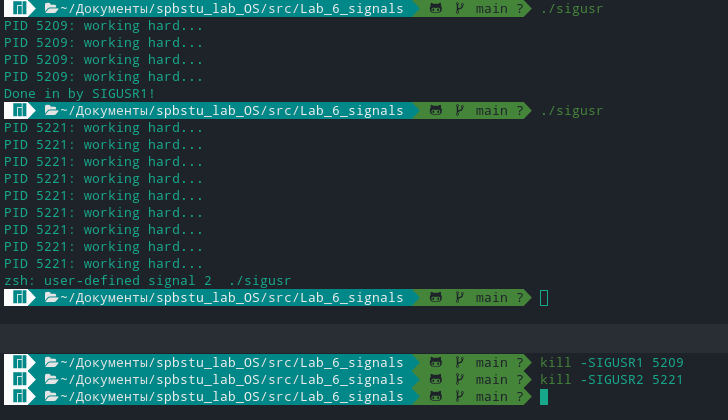
SIGKILL – убивает процесс

SIGINT – завершает процесс

SIGSTOP – приостанавливает процесс

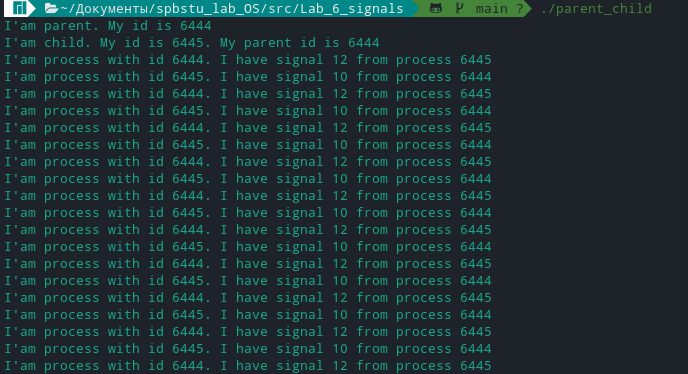
SIGCONT – возобновляет процесс в бэкграунде

SIGQUIT – остановка процесса + выполнение дампа памяти для процесса (срез данных непосредственно из самой памяти, из ОЗУ)



SIGUSR1 – этот сигнал ожидает программа и при получении его завершается сама

1. Составьте программу, запускающую процесс-потомок. Процессродитель и процесс-потомок должны генерировать (можно случайным образом) и отправлять друг другу сигналы (например, SIGUSR1, SIGUSR2). Каждый из процессов должен выводить на консоль информацию об отправленном и о полученном сигналах.
2. Для организации обработчиков сигналов предпочтительно использовать системный вызов sigaction() и соответствующую структуру данных. Обеспечьте корректное завершение процессов.



|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <errno.h>  #include <signal.h>  int process;  void sigusr1\_handler(int sig)  {  printf("I'am process with id %d. I have signal %d from process %d\n", getpid(), sig, process);  sleep(1);  }  int main()  {  int child = 0;  child = fork();  struct sigaction sa;  sa.sa\_handler = sigusr1\_handler;  sa.sa\_flags = 0;  sigemptyset(&sa.sa\_mask);  if ( child > 0)  {  if (sigaction(SIGUSR2, &sa, NULL) == -1) {  perror("sigaction");  exit(1);  }  process = child;  printf("I'am parent. My id is %d\n", getpid());  while(1) {  sleep(5);  kill(process, SIGUSR1);  }  }  else  {  if (sigaction(SIGUSR1, &sa, NULL) == -1) {  perror("sigaction");  exit(1);  }  process = getppid();  printf("I'am child. My id is %d. My parent id is %d\n", getpid(), getppid());  while(1) {  sleep(5);  kill(process, SIGUSR2);  }  }  return 0;  } |

Системный вызов signal() устанавливает новый обработчик сигнала с номером signum в соответствии с параметром sighandler, который может быть функцией пользователя, SIG\_IGN или SIG\_DFL  
  
Разница:

1. Функция signal() не обязательно блокирует поступление других сигналов во время выполнения текущего обработчика; функция sigaction() может блокировать другие сигналы до тех пор, пока текущий обработчик не вернется.
2. Функция signal() (обычно) сбрасывает действие сигнала обратно к SIG\_DFL (по умолчанию) почти для всех сигналов. Это означает, что обработчик signal() должен переустановить себя в качестве первого действия. Он также открывает окно уязвимости между временем обнаружения сигнала и переустановкой обработчика, в течение которого, если поступает второй экземпляр сигнала, происходит поведение по умолчанию (обычно завершается, иногда с предубеждением - он же дамп ядра).
3. sigaction() возвращает 0, если успешно и -1 в случае ошибки, signal() возвращает предыдущее значение обработчика сигнала или SIG\_ERR при ошибке.
4. Модифицируйте программу занятия 3 (файлы pipe\_server.cpp , pipe\_client.cpp и pipe\_local.h), сделав ее более стабильной в работе. В числе недостатков, которые желательно устранить, можно указать:
   * если клиентский процесс завершается по получению сигнала SIGINT (Ctrl+C), то private FIFO не удаляется из системы (исправляется посредством организации перехвата сигнала с выполнением необходимых действий);
   * клиентский процесс при его инициализации может обрушиться, если сервер окажется недоступен (исправляется путем попытки запуска сервера из клиента, если сервер не активен).

|  |
| --- |
| /\* The client program pipe\_client.cpp \*/  #include"pipe\_local.h"  FILE \*server;  void sigint\_handler(int sig) {  if (sig == SIGINT) {  exit(0);  }  }  int main(void)  {  int n, privatefifo, publicfifo;  static char buffer[PIPE\_BUF];  struct message msg;  /\* Make the name for the private FIFO \*/  sprintf(msg.fifo\_name, "/tmp/fifo %d", getpid());  struct sigaction sa;  sa.sa\_handler = sigint\_handler;  sa.sa\_flags = 0;  sigemptyset(&sa.sa\_mask);  if (sigaction(SIGINT, &sa, NULL) == -1) {  perror("sigaction");  exit(1);  }  /\* Generate the private FIFO \*/  if (mknod(msg.fifo\_name, S\_IFIFO | 0666, 0)<0){  perror(msg.fifo\_name);  exit(1);  }  /\* OPEN the public FIFO for writing \*/  int pid = 0;  if ((publicfifo=open(PUBLIC, O\_WRONLY))==-1){  pid = fork();  if (pid == 0) {  execl("./server", "&");  }  else {  sleep(2);  publicfifo=open(PUBLIC, O\_WRONLY);  }  }  while(1){ /\* FOREVER \*/  write(fileno(stdout), "\ncmd>", 6);  memset(msg.cmd\_line, 0x0, B\_SIZ); /\* Clear first \*/  n = read(fileno(stdin), msg.cmd\_line, B\_SIZ); /\* Get cmd \*/  if(!strncmp("quit", msg.cmd\_line, n-1))  break; /\* EXIT? \*/  write(publicfifo, (char \*) &msg, sizeof(msg)); /\* to PUBLIC \*/  /\* OPEN private FIFO to read returned command oputput \*/  if((privatefifo = open(msg.fifo\_name, O\_RDONLY))==-1){  perror(msg.fifo\_name);  exit(3);  }  /\* READ private FIFO and display on standard error \*/  while((n=read(privatefifo, buffer, PIPE\_BUF))>0){  write(fileno(stderr), buffer, n);  }  close(privatefifo);  }  close(publicfifo);  unlink(msg.fifo\_name);  } |

|  |
| --- |
| /\* The server program pipe\_server.cpp \*/  #include"pipe\_local.h"  int publicfifo;  void sigint\_handler(int sig) {  if (sig == SIGINT) {  close(publicfifo);  remove(PUBLIC);  exit(0);  }  }  int main(void)  {  int n, done, dummyfifo, privatefifo;  static char buffer[PIPE\_BUF];  FILE \*fin;  struct message msg;  /\* Generate the public FIFO \*/  mknod(PUBLIC, S\_IFIFO | 0666, 0);  struct sigaction sa;  sa.sa\_handler = sigint\_handler;  sa.sa\_flags = 0;  sigemptyset(&sa.sa\_mask);  if (sigaction(SIGINT, &sa, NULL) == -1) {  perror("sigaction");  exit(1);  }  /\* OPEN the public FIFO for reading and writing \*/  if ((publicfifo=open(PUBLIC, O\_RDONLY))==-1 ||  (dummyfifo=open(PUBLIC, O\_WRONLY | O\_NDELAY))==-1){  perror(PUBLIC);  exit(1);  }  /\* Message can be read from the PUBLIC pipe \*/  while(read(publicfifo, (char \*) &msg, sizeof(msg))>0){  n = done = 0; /\* Clear counters | flags \*/  do{ /\* Try OPEN of private FIFO \*/  if ((privatefifo=open(msg.fifo\_name, O\_WRONLY | O\_NDELAY))==-1)  sleep(3); /\* Sleep a while \*/  else{ /\* OPEN succesful \*/  fin = popen(msg.cmd\_line, "r"); /\* Execute the cmd \*/  write(privatefifo, "\n", 1); /\* Keep output pretty \*/  while((n=read(fileno(fin), buffer, PIPE\_BUF))>0){  write(privatefifo, buffer, n); /\*to private FIFO \*/  memset(buffer, 0x0, PIPE\_BUF); /\* Clear it out \*/  }  pclose(fin);  close(privatefifo);  done = 1; /\* Record succes \*/  }  }while(++n<5 && !done);    if(!done) /\* Indicate failure \*/  write(fileno(stderr), "\nNOTE: SERVER \*\* NEVER \*\* accessed private FIFO\n", 48);  }  return 0;  } |

**Вывод**

В ходе работы было освоены средства управления процессами, позволяющего процессам передавать информацию о каких-либо событиях,

отрабатывать реакции на различные события и взаимодействовать друг с другом.

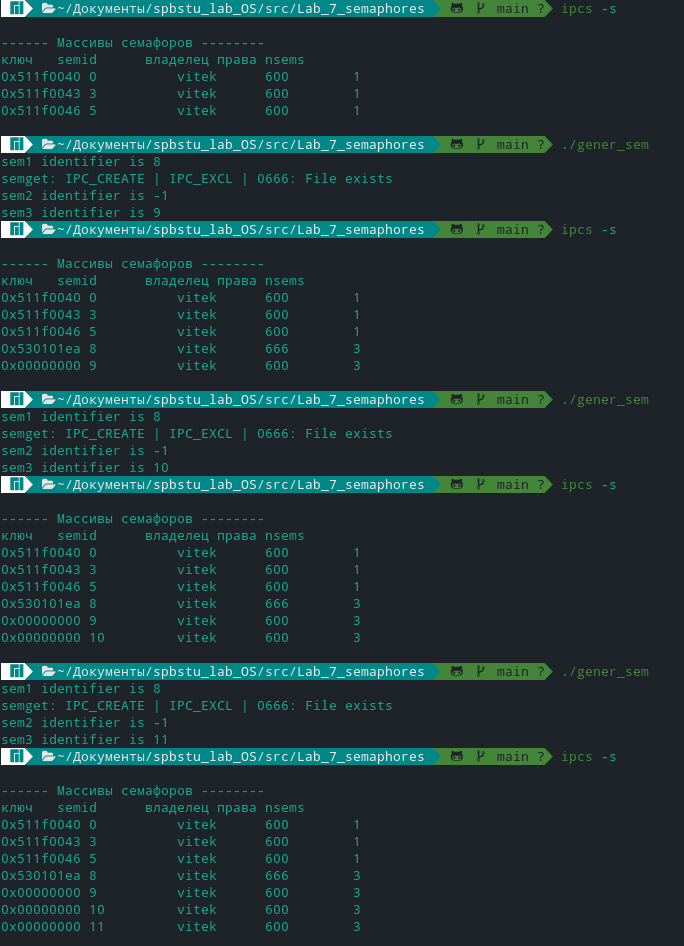
# **Лабораторная работа №7 «Семафоры и синхронизация»**

**Цель работы**

Освоение семафоров (semaphores) как эффективных средств синхронизации доступа процессов к разделяемым ресурсам операционной системы, а также синхронизации доступа потоков (в части 2) к разделяемым ресурсам процесса.

**Последовательность выполнения работы:**

1. Войдите в систему и скопируйте с разделяемого ресурса в свой HOME каталог набор исходных файлов для седьмого занятия.
2. 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() флагов.



Первый аргумент semget() - это ключ (в нашем случае возвращается ftok()-ом). Он сравнивается с ключами остальных множеств семафоров, присутствующих в системе. Вместе с этим решается вопрос о выборе между созданием и подключением к множеству семафоров в зависимости от аргумента msgflg.

ftok - преобразовывает имя файла и идентификатор проекта в ключ для системных вызовов. Системный вызов semget возвращает идентификатор множества семафоров, ассоциированный с ключом key.

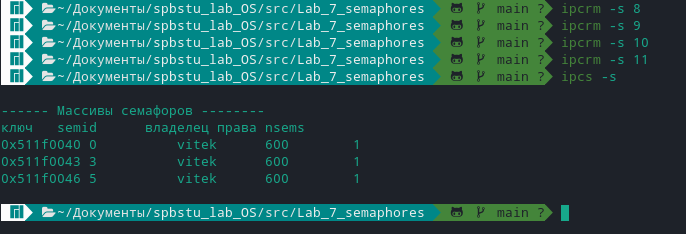
При запуске данной программы несколько раз видим, что:

• набор sem1 будет создан единожды, а затем каждая новая попытка будет всего лишь открывать доступ к существующему ресурсу;

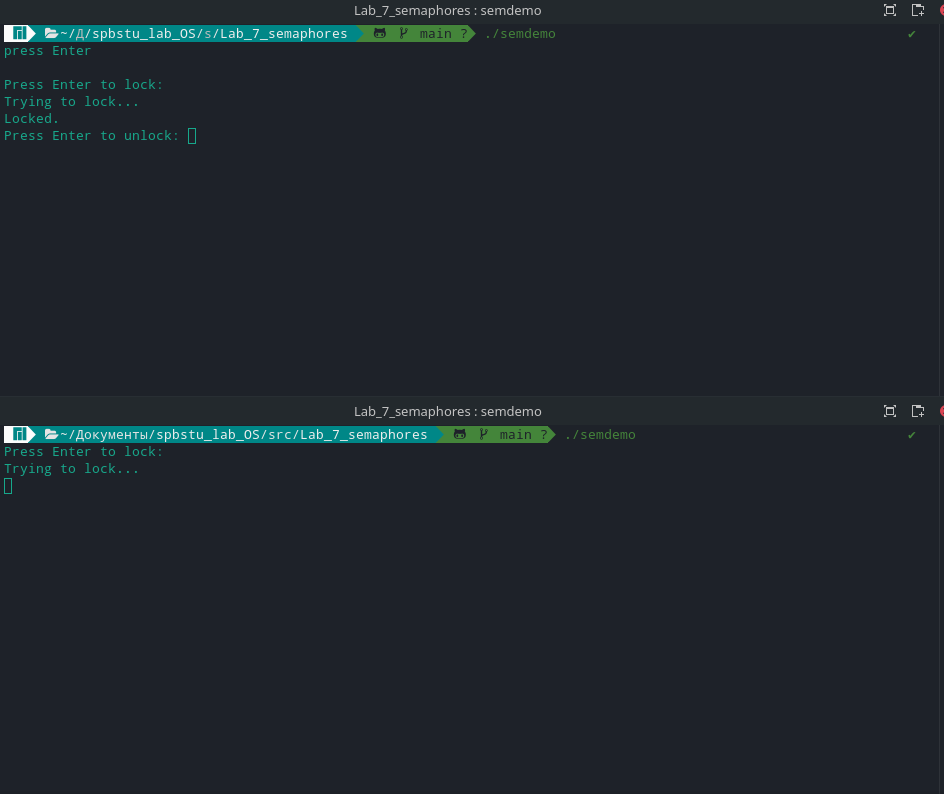
• попытки создания набора sem2 на том же ключе всегда будут приводить к ошибке из-за наличия флагов IPC\_CREATE | IPC\_EXCL, не допускающих открытия ресурса вместо его создания;

• набор sem3 будет создаваться при каждом новом запуске программы. Причем каждый раз с новым уникальным идентификатором.

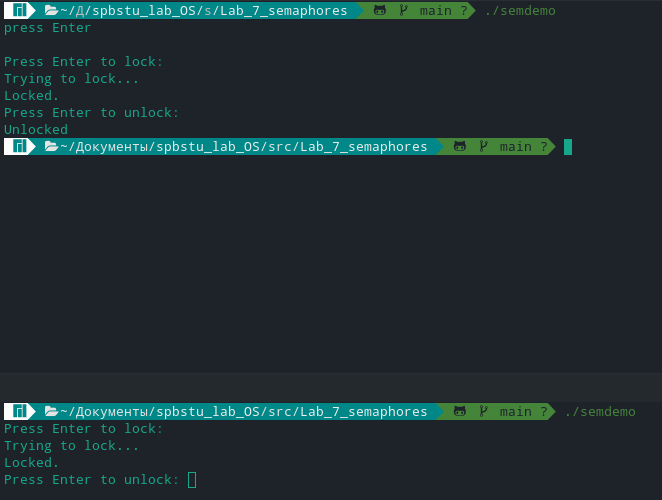
1. Удалите созданные на предыдущем шаге семафоры с помощью команды ipcrm с соответствующей опцией и значением id семафора или ключа.



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.

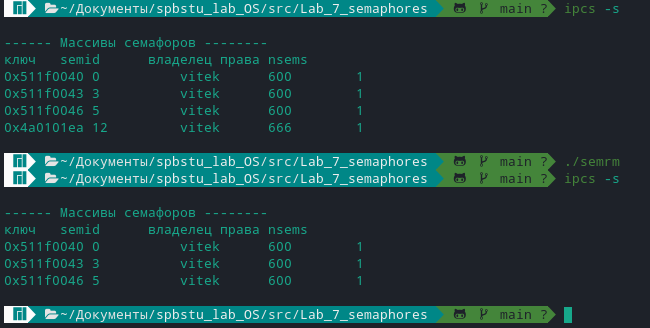


В верхнем терминале мы заблокировали ресурс, а в нижнем пытаемся заблокировать этот же ресурс, но вынуждены ждать, пока он освободится другим процессом (верхний терминал).

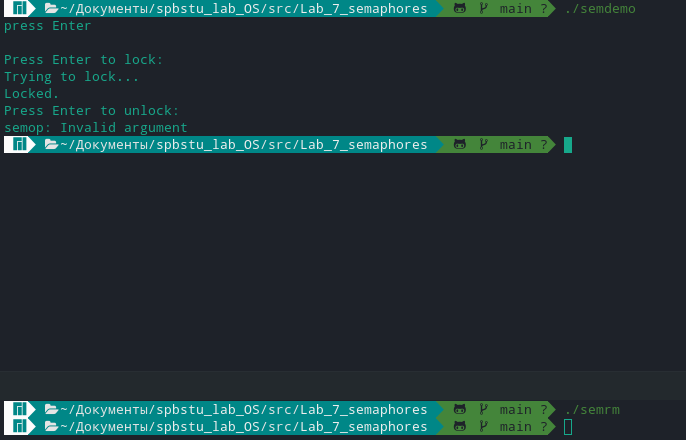


Как только мы освободили ресурс в верхнем терминале он тут же был занят ожидающим его нижним терминалом.

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 .



Программа может удалить только те семафоры, что были созданы при выполнении предыдущего пункта т. к. в semrn.cpp задается тот же ключ key = ftok(".", u\_char) (где u\_char = `J`; ключи семафоров уникальны)

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

Т.к. семафор с данным ключом посреди работы программы был удален, вызывается ошибка.

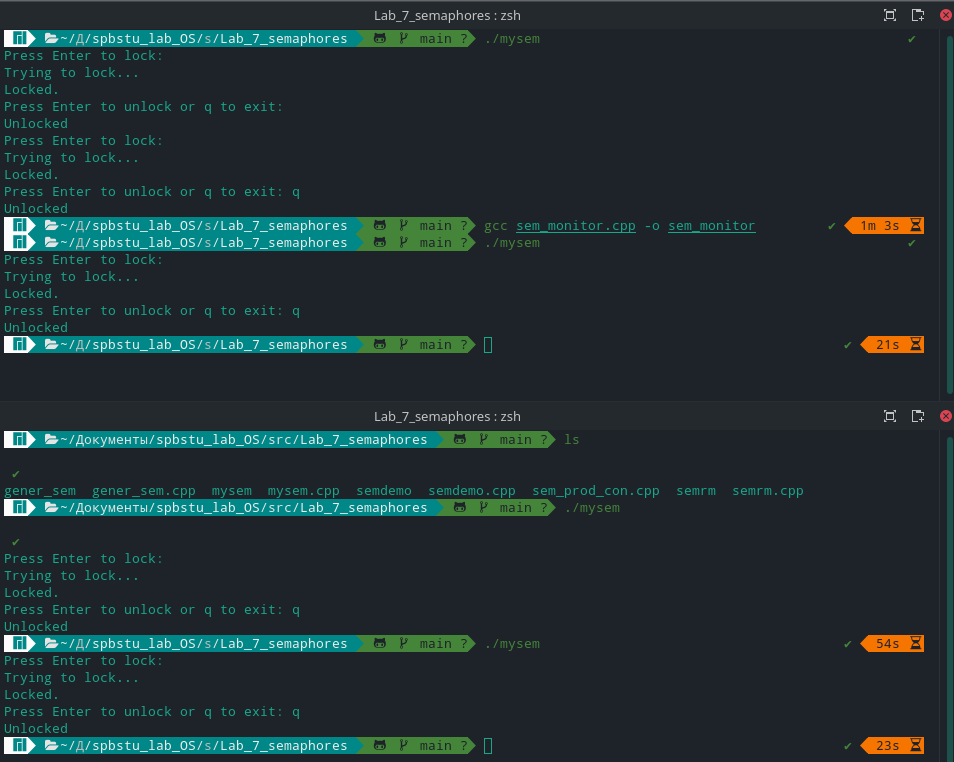
1. Попытайтесь улучшить программу semdemo.cpp , например, предоставив процессу возможность после освобождения ресурса становиться снова в очередь на повторное его занятие (а не завершаться), организовав при этом завершение процесса по вводу какого-либо символа.

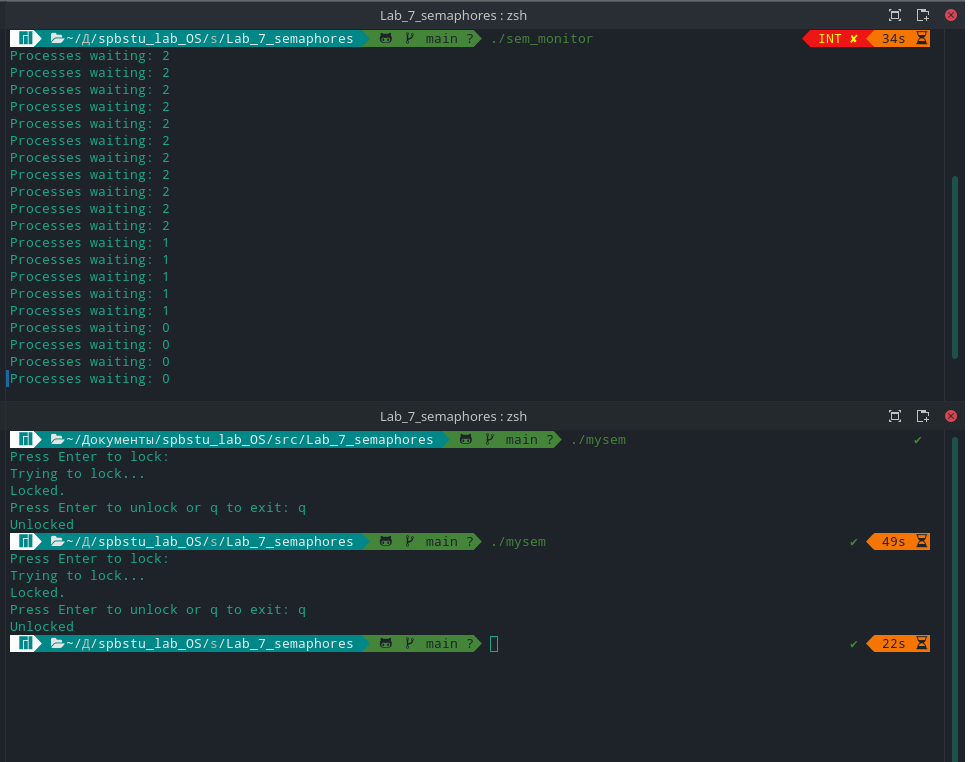


|  |
| --- |
| int main(void)  {  int isWorking = 1;  key\_t key;  int semid;  char u\_char = 'J';  struct sembuf sb;  while (isWorking) {  sb.sem\_num = 0;  sb.sem\_op = -1; /\* set to allocate resource \*/  sb.sem\_flg = SEM\_UNDO;  if ((key = ftok(".", u\_char)) == -1) {  perror("ftok");  exit(1);  }  /\* grab the semaphore set created by initsem: \*/  if ((semid = initsem(key, 1)) == -1) {  perror("initsem");  exit(1);  }  printf("Press Enter to lock: ");  getchar();  printf("Trying to lock...\n");  if (semop(semid, &sb, 1) == -1) {  perror("semop");  exit(1);  }  printf("Locked.\n");  printf("Press Enter to unlock or q to exit: ");  if (getchar() == 'q') isWorking = 0;  sb.sem\_op = 1; /\* free resource \*/  if (semop(semid, &sb, 1) == -1) {  perror("semop");  exit(1);  }  printf("Unlocked\n");  }  return 0;  } |

1. Составьте программу, позволяющую мониторить количество процессов (типа semdemo), находящихся в состоянии ожидания освобождения ресурса (Trying to lock...) в каждый момент времени.

Программа строится на основе вызова semctl() с соответствующими параметрами и запускается на отдельном терминале.





|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <errno.h>  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h>  #define MAX\_RETRIES 10  union semun {  int val;  struct semid\_ds \*buf;  ushort \*array;  };  int initsem(key\_t key, int nsems) /\* key from ftok() \*/  {  int i;  union semun arg;  struct semid\_ds buf;  struct sembuf sb;  int semid;  semid = semget(key, nsems, IPC\_CREAT | IPC\_EXCL | 0666);  if (semid >= 0) { /\* we got it first \*/  sb.sem\_op = 1; sb.sem\_flg = 0;  arg.val = 1;  printf("press Enter\n"); getchar();  for(sb.sem\_num = 0; sb.sem\_num < nsems; sb.sem\_num++) {  /\* do a semop() to "free" the semaphores. \*/  /\* this sets the sem\_otime field, as needed below. \*/  if (semop(semid, &sb, 1) == -1) {  int e = errno;  semctl(semid, 0, IPC\_RMID); /\* clean up \*/  errno = e;  return -1; /\* error, check errno \*/  }  }  } else if (errno == EEXIST) { /\* someone else got it first \*/  int ready = 0;  semid = semget(key, nsems, 0); /\* get the id \*/  if (semid < 0) return semid; /\* error, check errno \*/  /\* wait for other process to initialize the semaphore: \*/  arg.buf = &buf;  for(i = 0; i < MAX\_RETRIES && !ready; i++) {  semctl(semid, nsems-1, IPC\_STAT, arg);  if (arg.buf->sem\_otime != 0) {  ready = 1;  } else {  sleep(1);  }  }  if (!ready) {  errno = ETIME;  return -1;  }  } else {  return semid; /\* error, check errno \*/  }  return semid;  }  int main(void)  {  key\_t key;  int semid;  char u\_char = 'J';  if ((key = ftok(".", u\_char)) == -1) {  perror("ftok");  exit(1);  }  /\* grab the semaphore set created by initsem: \*/  if ((semid = initsem(key, 1)) == -1) {  perror("initsem");  exit(1);  }  while (true){  int output = semctl(semid, 0, GETNCNT);  printf("Processes waiting: %d\n", output);  sleep(1);  }  return 0;  } |

Вывод

Для синхронизации доступа к разделяемым ресурсам удобно использовать семафоры. Мы научились методам их создания, мониторинга, удаления, а также работы с ними.

# **Лабораторная работа №8 «Обмен через очереди сообщений»**

**1. Цели работы**

Знакомство с возможностями очередей сообщений (Message Queues) – мощного и гибкого средства межпроцессорного взаимодействия в OS Linux

**2. Задачи**

1. Войти в систему и скопировать в свой HOME-каталог с разделяемого ресурса набор исходных файлов для восьмой лабораторной работы.

2. Скомпилировать и выполнить программу gener\_mq.cpp, которая создает несколько очередей сообщений. После завершения программы выполнить программу ipcs и привести отличия результата от того, что был при вызове подобной команды из программы.

3. Скомпилировать программы sender.cpp и receiver.cpp, задав соответствующим исполняемым файлам разные имена (g++ <имя.cpp файла> -o <имя файла>). Запустить процессы на разных терминалах и необходимо передать им текстовые сообщения от процесса sender процессу receiver. Проанализировать, что происходит с ресурсом Message Queue, после завершения каждого из процессов (командой ipcs). При этом выполнить различные виды завершения отправкой сигналов SIGQUIT и SIGINT (нажатием Ctrl+C).

4. Запустить несколько процессов receiver на различных терминалах и, отправляя сообщения процессом sender, проанализировать ситуацию.

5. Модифицировать программы sender.cpp и receiver.cpp так, чтобы организовать отправку сообщений двух типов через одну и ту же очередь для двух различных процессов получателей. Для этого необходимо управлять параметром поля mtype структуры my\_msgbuf на передающей стороне и параметром msgtyp в системном вызове msgrcv() на приемной стороне.

**3. Ход выполнения работы**

1. Войду в систему и скопирую в свой HOME-каталог с разделяемого ресурса набор исходных файлов для второй лабораторной работы. Содержимое файлов приведено в листингах 8.1 – 8.3, соответственно.

|  |
| --- |
| Листинг 8.1 Содержимое файла gener\_mq.cpp |
| /\* The program gener\_mq.cpp \*/  /\* Message queue generation \*/  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <limits.h>  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #define MAX 5  main(void){  FILE \*fin;  char buffer[PIPE\_BUF];  char u\_char = 'A';  int i, n, mid[MAX];  key\_t key;  for (i=0; i<MAX ; ++i, ++u\_char){  key = ftok(".", u\_char);  if ((mid[i] = msgget(key, IPC\_CREAT | 0660))==-1){  perror("Queue create");  exit(1);  }  }  fin = popen("ipcs", "r"); /\* Run the ipcs command \*/  while((n = read(fileno(fin), buffer, PIPE\_BUF))>0)  write(fileno(stdout), buffer, n);  /\*Display the ipcs output \*/  pclose(fin);  for (i=0; i<MAX; ++i)  msgctl(mid[i], IPC\_RMID, NULL);  /\* remove \*/  exit(0);  } |

|  |
| --- |
| Листинг 8.2 Содержимое файла reciever.cpp |
| /\*  \*\* receiver.cpp -- reads from a message queue  \*/  #include <stdio.h>  #include <stdlib.h>  #include <errno.h>  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  struct my\_msgbuf {  long mtype;  char mtext[200];  };  int main(void)  {  struct my\_msgbuf buf;  int msqid;  key\_t key;  if ((key = ftok(".", 'B')) == -1) { /\* same key as sender.cpp \*/  perror("ftok");  exit(1);  }  if ((msqid = msgget(key, 0644)) == -1) { /\* connect to the queue \*/  perror("msgget");  exit(1);  }  printf("spock: ready to receive messages, captain.\n");  for(;;) { /\* Spock never quits! \*/  if (msgrcv(msqid, &buf, sizeof(buf.mtext), 0, 0) == -1) {  perror("msgrcv");  exit(1);  }  printf("spock: \"%s\"\n", buf.mtext);  }  return 0;  } |

|  |
| --- |
| Листинг 8.3 Содержимое файла sender.cpp |
| /\*  \*\* sender.cpp -- writes to a message queue  \*/  #include <stdio.h>  #include <stdlib.h>  #include <errno.h>  #include <string.h>  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  struct my\_msgbuf {  long mtype;  char mtext[200];  };  int main(void)  {  struct my\_msgbuf buf;  int msqid;  key\_t key  if ((key = ftok(".", 'B')) == -1) {  perror("ftok");  exit(1);  }  if ((msqid = msgget(key, 0644 | IPC\_CREAT)) == -1) {  perror("msgget");  exit(1);  }  printf("Enter lines of text, ^D to quit:\n");  buf.mtype = 1; /\* we don't really care in this case \*/  while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {  int len = strlen(buf.mtext);  /\* ditch newline at end, if it exists \*/  if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';  if (msgsnd(msqid, &buf, len+1, 0) == -1) /\* +1 for '\0' \*/  perror("msgsnd");  }  if (msgctl(msqid, IPC\_RMID, NULL) == -1) {  perror("msgctl");  exit(1);  }  return 0;  } |

2. Скомпилирую и выполню программу gener\_mq.cpp, которая создает несколько очередей сообщений. С помощью опции g++ выполню компиляцию. Полученный результат программы gener\_mq.cpp записывается в файл gener\_mq. Выполню его в терминале. Вывод в терминал представлен на рисунке 8.1. После завершения программы выполню команду ipcs. Результат приведен на рисунке 8.2. Важно отметить, что при исполнении программы gener\_mq создается 5 очередей сообщений и после выводится информация о механизмах IPC, а далее очереди сообщений удаляются, но при вызове команды ipcs выводится информация о всех механизмах в системе IPC, но при этом очереди не создаются.

|  |
| --- |
| Рис. 8.1 – Выполнение программы gener\_mq |
| Рис. 8.2 – Выполнение команды ipcs |

3. Скомпилирую программы sender.cpp и receiver.cpp, задав соответствующим исполняемым файлам разные имена (g++ <имя.cpp файла> -o <имя файла>). Выполнение представлено на рисунке 8.3.

|  |
| --- |
| Рис. 8.3 – Компиляция файлов |

Далее запущу процессы на разных терминалах и передам текстовые сообщения от процесса sender процессу receiver. Иллюстрация выполненных действия представлена на рисунке 8.4.

|  |
| --- |
| Рис. 8.4 - Передача текстовых сообщений между процессами |

При завершении передачи Ctrl+D, очередь сообщений удаляется. Результаты приведены на рисунке 8.5.

|  |
| --- |
| Рис. 8.5 – Завершение передачи сообщений через Ctrl+D |

Важно отметить, что данные программы обмениваются сообщениями только используя очередь сообщений, что представлено на рисунке 8.4. Прерывание процессов с помощью SIGQUIT представлено на рисунке 8.6. В данном случаем очередь сообщений не удаляется и остается в системе. Аналогично и SIGINT. Результаты приведены на рисунке 8.7.

|  |
| --- |
| Рис. 8.6 – Завершение с помощью SIGQUIT |
| Рис. 8.7 – Завершение с помощью SIGINT |

4. Необходимо исследовать ситуацию, когда процесс receiver запускается уже после того, как процесс sender отправил в очередь одно или множество сообщений. Для этого выполню файл sender, отправлю несколько сообщений, а после уже создам процесс receiver. Результаты приведены на рисунке 8.8. Из результатов видно, что созданный позже процесс receiver получил сообщения, так как сообщения будут ждать в очереди, даже если и sender завершиться c помощью SIGINT или SIGQUIT. Моделирование ситуации, когда sender завершается раньше приведено на рисунке 8.9.

|  |
| --- |
| Рис. 8.8 – Запуск receiver позже отправки сообщений |
| Рис. 8.9 – Запуск receiver при завершенном sender |

5. Запущу несколько процессов receiver на различных терминалах и, отправляя сообщения процессом sender, посмотрю, что будет при отправке сообщений. Моделирование ситуации представлено на рисунке 8.10.

|  |
| --- |
| Рис. 8.10 – Запуск двух процессов receiver в двух разных терминалах |

Чтобы протестировать ситуацию, введу четыре разных сообщения. Результаты приведены на рисунке 8.11. Как можно увидеть, сообщения доходят не все, а только половина. Данный факт обусловлен тем, что сообщение читается и сразу же удаляется из очереди – другой процесс не может его прочитать, а вот во время получения первого, второй успевает прочитать второе сообщение и так далее.

|  |
| --- |
| Рис. 8.11 – Результаты тестирования отправки на два процесса |

6. Необходимо модифицировать программы sender.cpp и receiver.cpp так, чтобы была организована отправка сообщений двух типов через одну и ту же очередь для двух 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. Модифицированные программы представлены в листингах 8.4 и 8.5 соответственно.

|  |
| --- |
| Листинг 8.4 – Модифицированная программа sender\_m.cpp |
| /\*  \*\* sender.cpp -- writes to a message queue  \*/  #include <stdio.h>  #include <stdlib.h>  #include <errno.h>  #include <string.h>  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  struct my\_msgbuf {  long mtype;  char mtext[200];  };  int main(void)  {  struct my\_msgbuf buf;  int msqid;  key\_t key  if ((key = ftok(".", 'B')) == -1) {  perror("ftok");  exit(1);  }  if ((msqid = msgget(key, 0644 | IPC\_CREAT)) == -1) {  perror("msgget");  exit(1);  }  printf("Enter lines of text, ^D to quit:\n");  buf.mtype = 1; /\* we don't really care in this case \*/  while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {  int len = strlen(buf.mtext);  /\* ditch newline at end, if it exists \*/  if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';  for(long i = 1; i <= 2; i++){  buf.mtype = i;  if (msgsnd(msqid, &buf, len+1, 0) == -1) /\* +1 for '\0' \*/  perror("msgsnd");  }  }  if (msgctl(msqid, IPC\_RMID, NULL) == -1) {  perror("msgctl");  exit(1);  }  return 0;  } |

|  |
| --- |
| Листинг 8.5 – Модифицированная программа receiver\_m.cpp |
| /\*  \*\* receiver.cpp -- reads from a message queue  \*/  #include <stdio.h>  #include <stdlib.h>  #include <errno.h>  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  struct my\_msgbuf {  long mtype;  char mtext[200];  };  int main(void)  {  struct my\_msgbuf buf;  int msqid;  key\_t key;  if ((key = ftok(".", 'B')) == -1) { /\* same key as sender.cpp \*/  perror("ftok");  exit(1);  }  if ((msqid = msgget(key, 0644)) == -1) { /\* connect to the queue \*/  perror("msgget");  exit(1);  }  printf("spock: ready to receive messages, captain.\n");  for(;;) { /\* Spock never quits! \*/  if (msgrcv(msqid, &buf, sizeof(buf.mtext), buf.mtype, 0) == -1) {  perror("msgrcv");  exit(1);  }  printf("spock: \"%s\"\n", buf.mtext);  }  return 0;  } |

Результаты отправки сообщений на два процесса получателя приведены на рисунке 8.12. Как видно сообщений отправляются корректно. Важно отметить, чтобы изменять количество процессов-приемников нужно изменить for() в sender\_m.cpp.

|  |
| --- |
| Рис. 8.12 – Моделирование отправки на два процесса-получателя |

**4. Вывод**

В ходе выполнения работы были изучены основы межпроцессорного взаимодействия в ОС Linux при помощи очередей сообщений между процессами. При помощи параметров была настроена передача сообщений одновременно нескольким процессам-получателям.

**Лабораторная работа №9 «Работа с разделяемой памятью»**

**1. Цели работы**

Использование для обмена данными разделяемой памяти (shared memory) - самого быстрого средства межпроцессорного взаимодействия в Linux.

**2. Задачи**

1. Войти в систему и скопировать с разделяемого ресурса в свой HOME-каталог набор исходных файлов для десятого занятия.

2. Скомпилировать и выполнить программу gener\_shm.cpp, которая демонстрирует создание сегментов разделяемой памяти. Запустить программу несколько раз и выполнить команду ipcs -m.

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

4. Скомпилировать 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.

5. Запустить несколько раз процессы типа shmdemo с различными значениями параметров и проиллюстрируйте возможности чтения и записи в сегмент общей памяти независимо исполняемыми процессами. Затем удалить сегмент памяти командой ipcrm.

6. Скомпилировать и выполните программу attach\_shm.cpp, иллюстрирующую передачу символьной информации между двумя процессами (родственными) через сегмент общей памяти с модификацией этой информации.

7. Составить программу, создающую три разделяемых сегмента памяти размером 1023 байта каждый. Укажите в вызове shmat() параметр shmaddr = 0 при привязке сегментов.

**3. Ход выполнения работы**

1. Войду в систему и скопировать с разделяемого ресурса в свой HOME-каталог набор исходных файлов для десятого занятия.

2. Скомпилирую и выполню программу gener\_shm.cpp, которая демонстрирует создание сегментов разделяемой памяти. Запустить программу несколько раз и выполнить команду ipcs -m. Результаты приведены на рисунке 9.1 и 9.2. Видно, что создается два сегмента с id 22 и 23, а во втором случае с 22 и 24. Важно уточнить, что первый участок создается по идентификатору 22, а после системный вызов возвращает его идентификатор. Второй участок создается каждый раз, так как его значение key равно IPC\_PRIVATE.

|  |
| --- |
| Рис. 9.1 Результат выполнения программы gener\_shm.cpp |

3. Удалю созданные на предыдущем шаге сегменты разделяемой памяти с помощью команды ipcrm с соответствующей опцией и значением id сегмента или ключа. Результаты представлены на рисунке 9.2.

|  |
| --- |
| Рис. 9.2 – Удаление разделяемой памяти |

4. Скомпилирую 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. Важно отметить, что программа создает участок памяти равный 1024 байтам. Выделенный участок памяти представлен на рисунке 9.3 под id равным 42.

|  |
| --- |
| Рис. 9.3 – Создание участка памяти |

5. Запущу несколько раз процессы типа shmdemo с различными значениями параметров и проиллюстрирую возможности чтения и записи в сегмент общей памяти независимо исполняемыми процессами. Затем удалю сегмент памяти командой ipcrm. Как видно из полученных изображений сначала программа получает доступ к разделяемому участку памяти по ключу. После проходит вызов shamat(), который подключается к памяти с полученным идентификатором. После введение строки, программа или записывает или выводит строку. На рисунке 9.4 приведено записывание строки в разделяемую память. В завершении удаляю сегмент выделенной памяти (рисунок 9.5)

|  |
| --- |
| Рис. 9.4 – Запись строки в память |
| 9.5 – Удаление выделенного сегмента |

6. Скомпилирую и выполню программу attach\_shm.cpp, иллюстрирующую передачу символьной информации между двумя процессами (родственными) через сегмент общей памяти с модификацией этой информации. Как видно из полученных данных, приведенных на рисунке 9.6, в сегмент памяти записывается английский алфавит, после процесс потомок изменяет ее, а именно изменяет регистр на нижний. Процесс родитель выводит измененные данные на экран и завершает работу. В программе происходит вызов shmdt(), который отделяет сегменты разделяемой памяти от процессов. После завершения процесса потомка, родитель помечает сегмент как удаленный с помощью вызова shmctl() и пользователь уже не может к нему обратиться.

|  |
| --- |
| Рис. 9.6 – Выполнение программы attach |

7. Составлю программу, создающую три разделяемых сегмента памяти размером 1023 байта каждый. Укажите в вызове shmat() параметр shmaddr = 0 при привязке сегментов. Важно отметить, что при обращении к 1024 сегменту памяти происходит ошибка. А также сегменты расположились по соседству, что и следовало проверить в задании. Результаты выполнения программы приведены на рисунке 9.7. Выделенные сегменты обозначены id равными 41, 42, 43.

|  |
| --- |
| Рис. 9.7 – Выполнение программы lab |

**4. Вывод**

В ходе выполнения программы было на практике изучено выделение и работа с разделяемой памятью. Были созданы сегменты, также производилась запись и чтение на сегменты разделяемой памяти. Был продемонстрирован процесс передачи информации между двумя процессами через сегмент памяти.

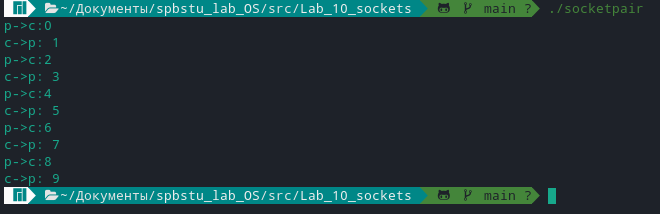
# **Лабораторная работа №10 «Создание соединений на сокетах»**

**Цель работы**

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

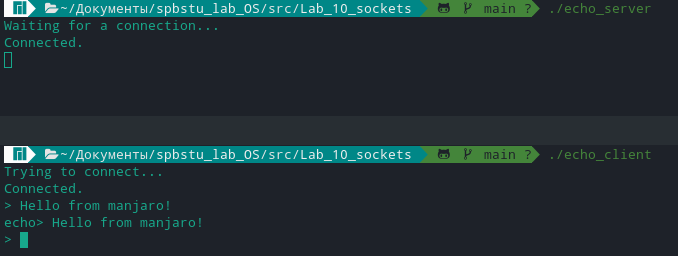
**Последовательность выполнения работы:**

1. Войдите в систему и скопируйте с разделяемого ресурса в свой HOME-каталог набор исходных файлов для десятого занятия.
2. 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 (в вызовах sleep()) в процессе-родителе и в процессе-потомке?



Родитель и потомок передают друг другу поочерёдно числа от 0 до 9, передача происходит через сокет. Зависимость от задержек есть только в скорости вывода, но порядок чисел изменён не будет.

1. 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. 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?

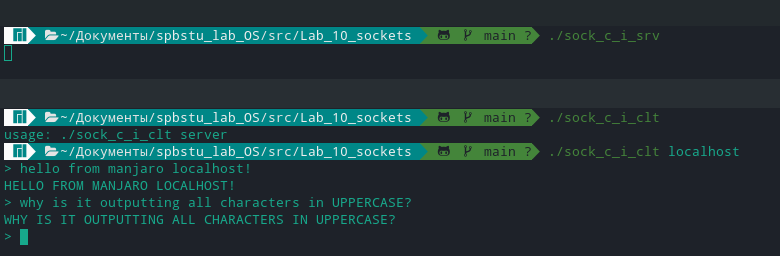


Видим, что результатом работы является вывод сообщения клиента клиенту только с добавлением “echo>” перед сообщением.

Тип сокета SOCK\_STREAM (двусторонние надежные последовательные потоки байтов) и адресное пространство AF\_UNIX (сокет рассчитан локальное соединение).

Создание специального файла связанно с созданием сокета в echo\_server.cpp. Сокет привязывается к UNIX адресу и после выполнения функции bind создается специальный файл по заданному пути.

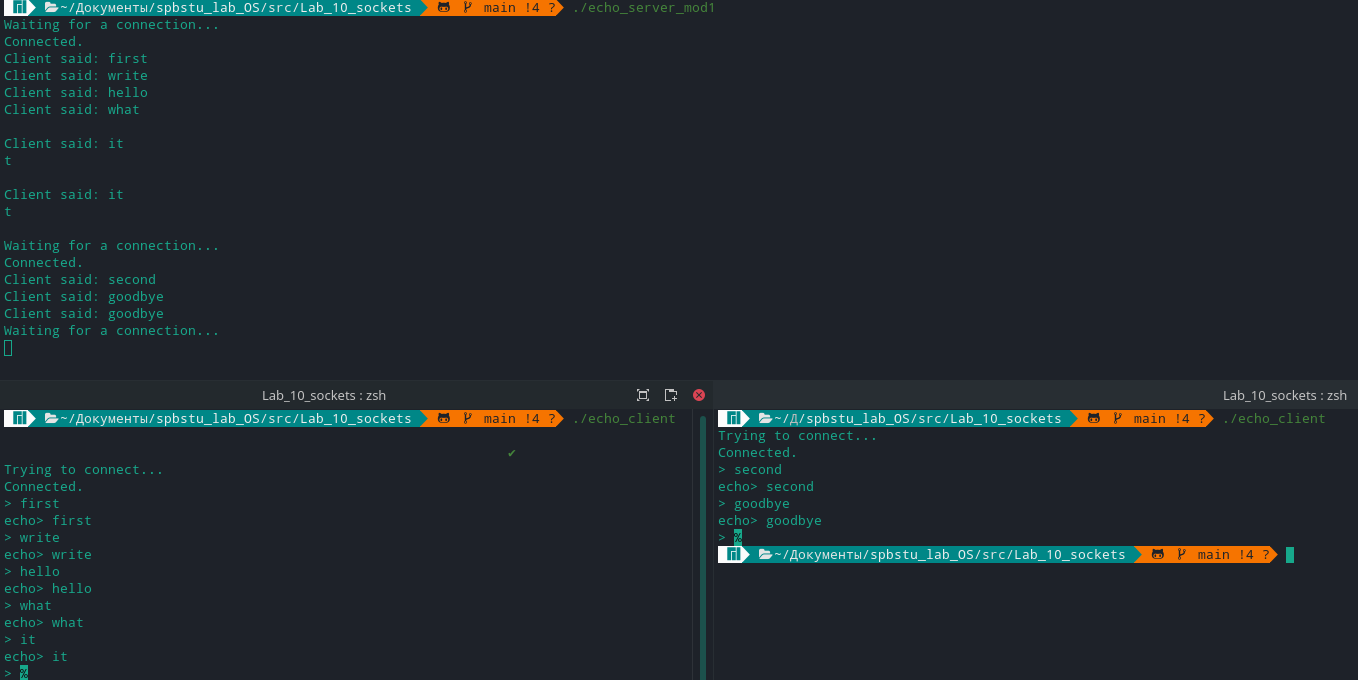
1. Скомпилируйте c разными именами программы sock\_c\_i\_srv.cpp и sock\_c\_i\_clt.cpp (в них используется общий include файл local\_c\_i.h). Запустите программы сервера и клиента на разных терминалах. При запуске клиента указывайте в качестве параметра командной строки имя хоста localhost. Введите символьную информацию в окне клиента и поясните вывод. Какой разновидности принадлежат сокеты, используемые в данном примере клиент-серверного взаимодействия?



Тип SOCK\_STREAM (двусторонние надежные последовательные потоки байтов) и адресное пространство AF\_INET (то есть сокет рассчитан на соединение по сети).

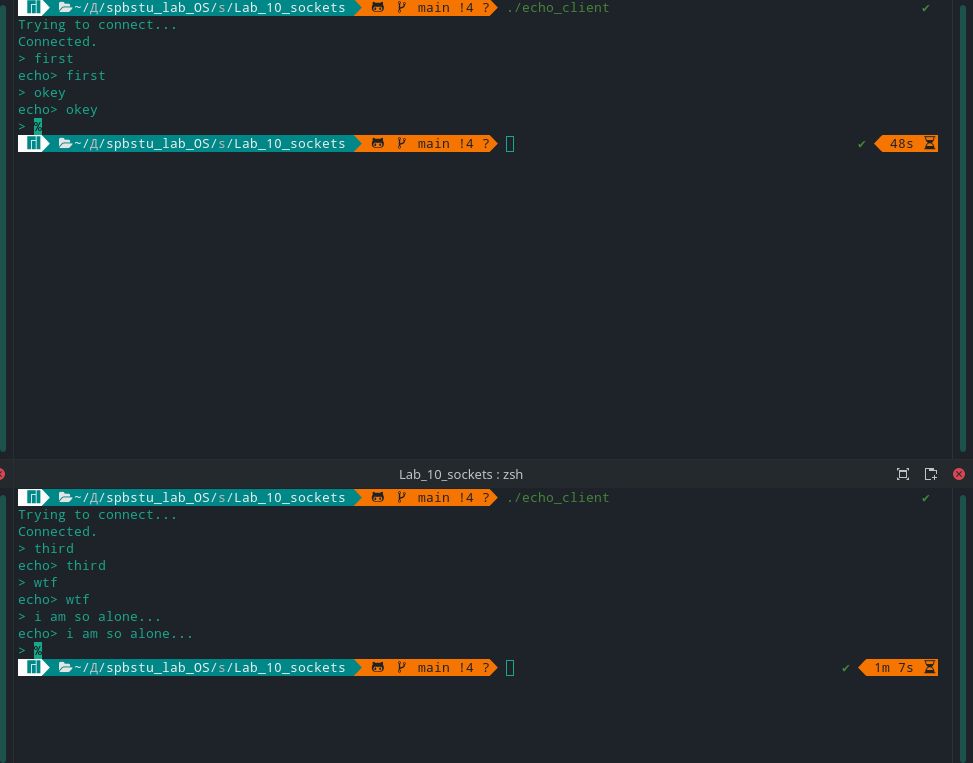
1. Модифицируйте программу echo\_server.cpp так, чтобы при ответе на запросы клиента что-либо выводилось в окне сервера. Испытайте работу эхосервера при одновременной работе с несколькими клиентами. Для вывода сообщения сервера просто попросим делать printf перед отправкой сообщения.

При одновременном подключении нескольких клиентов общается с сервером только первый. Остальные ждут. Как только первый отключается, подключается следующий.



Программа была доработана для работы с несколькими клиентами одновременно (происходит вызов fork и каждый клиент работает с новым дочерним процессом).





**Вывод**

В ходе лабораторной работы освоен набор системных вызовов для создания сокетных соединений различных типов для обмена данными по сети.