БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет прикладной математики и информатики

Кафедра технологий программирования

#### ОТЧЕТ

#### по лабораторной работе №3

По курсу “Программирование мобильных и встраиваемых систем”

***Командная оболочка bash: перенаправление ввода-вывода, сценарии и другие возможности***

**Подготовил:**  
студент 3 курса 13 группы кафедры ТП  
Петров Андрей Александрович

## Минск, 2021

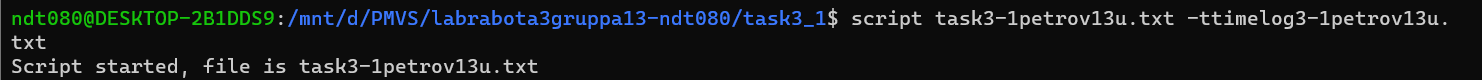
## ЦЕЛИ РАБОТЫ:

## Изучение файловой системы ОС Unix/Linux и основных функций для работы с каталогами и файлами; исследование методов создания процессов, основных функций создания и управления процессами, обмена данными между процессами.

## ЗАДАНИЕ 3.1. УПРАВЛЕНИЕ ПРОЦЕССАМИ

Изучите примеры задания 3.1 и выполните их в ОС Raspberry PI.

Включите ведение протокола командой script с журналом меток времени. **Протокол** назвать по следующему шаблону — taskXФамилияNM, где X — номер выполняемого задания, Фамилия — заменить на вашу фамилию латиницей и строчными буквами, N — номер группы, например 12 или 13, M — **r** — для Raspberry PI. **Журнал меток** назвать по следующему шаблону — timelogXФамилияNM, где X — номер выполняемого задания, Фамилия — заменить на вашу фамилию латиницей и строчными буквами, N — номер группы, например 12 или 13, M — **r** — для Raspberry PI.



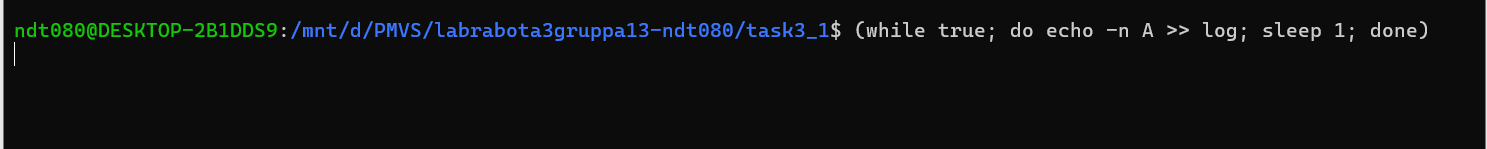
* + - 1. Используйте учетную запись, созданную в одной из предыдущих лабораторных работ. Войдите в систему на виртуальных терминалах 1 и 2 (tty1, tty2) под вашей учетной записью.

Изображение выглядит как текст, монитор, снимок экрана, черный

Автоматически созданное описание

* + - 1. Переключитесь в терминал tty1 и запустить процесс, выполняющий следующие команды:

*$ (while true; do echo -n A >> log; sleep 1; done)*



* + - 1. Заметьте, что сейчас этот терминал занят исполнением запущенного процесса, который исполняется на переднем плане. Этот процесс присоединяет символ "А"к файлу ~/log через каждую секунду. Чтобы визуально проверить это, переключитесь в виртуальный терминал tty2 и выполните следующую команду:

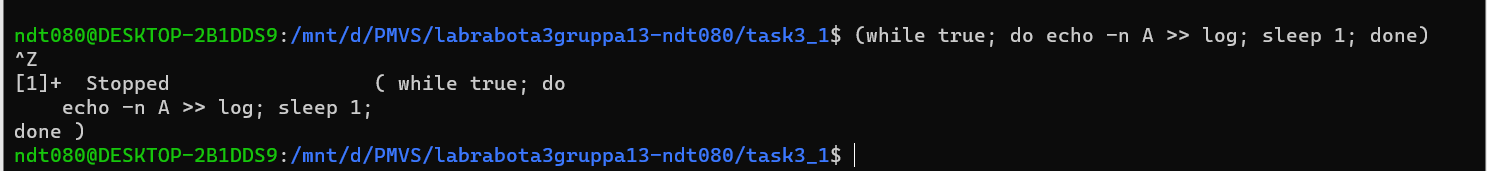
*$ tail -f log*

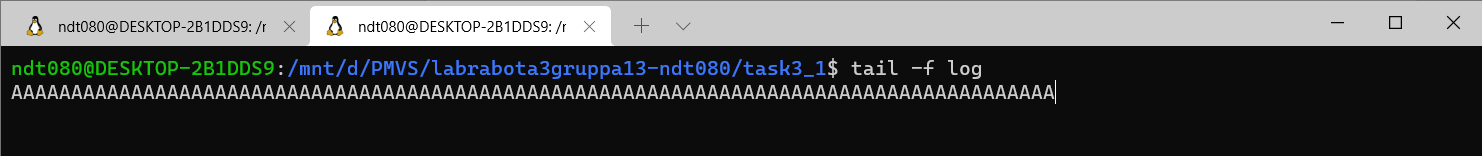
Вы должны увидеть последовательность символов, длина которой возрастает.

Изображение выглядит как текст

Автоматически созданное описание

* + - 1. Переключитесь в виртуальный терминал tty1 и приостановите работающий процесс, нажав клавиши <Ctrl+z>. Командная оболочка сообщит, что процесс остановлен и выдаст вам номер задания [1]. Переключитесь в виртуальный терминал tty2 и визуально проверьте, что файл ~/log больше не увеличивается.





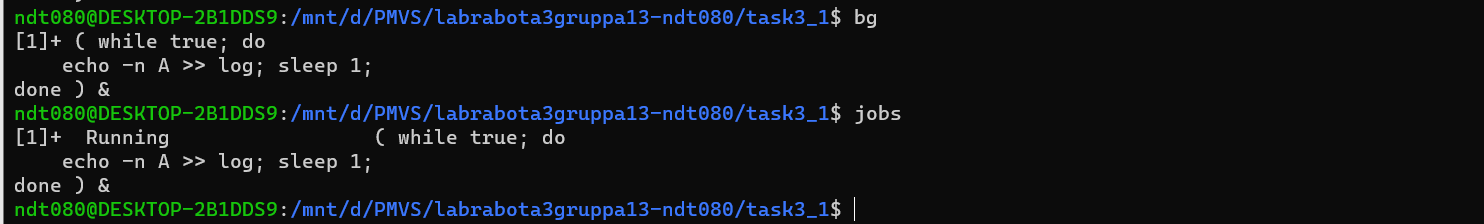
Строка в файле log перестала увеличиваться.

* + - 1. Переключитесь в виртуальный терминал tty1 и возобновите работу процесса в фоновом режиме. Используйте команду jobs, чтобы проверить, что задание [1] снова работает.

*$ bg*

*$ jobs*

Переключитесь в виртуальный терминал tty2 и визуально проверьте, что файл ~/log снова увеличивается.



Изображение выглядит как текст

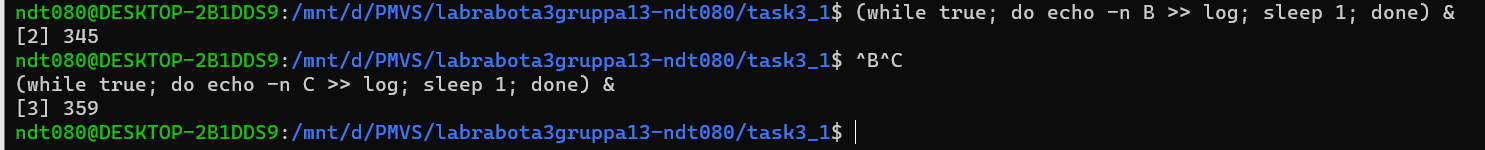
Автоматически созданное описание

* + - 1. Переключитесь в виртуальный терминал tty1 и запустите еще два процесса, выполнив следующие команды:

$ (while true; do echo -n B >> log; sleep 1; done) &

$ ^B^C

Вторая команда просто запускает предыдущую команду, заменяя символ "B"символом "C".



* + - 1. Выполните команду jobs и проверьте, что все три процесса работают. Переключитесь в виртуальный терминал tty2 и визуально проверьте, что файл ~/log снова увеличивается путем добавления символов "A" "B" и "C" через каждую секунду.

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

* + - 1. В пункте 4 вы приостановили исполнение процесса переднего плана путем нажатия клавиш <Ctrl+z>. В действительности эта комбинация нажатых клавиш посылает процессу сигнал. Используйте команду kill, чтобы получить список сигналов и соответствующие им имена и номера. Затем выполните команду kill, послав сигнал SIGSTOP заданию [1], чтобы приостановить его работу. Переключитесь в виртуальный терминал tty1 и выполните следующие команды:

*$ kill -l*

*$ kill -19 %1*

Изображение выглядит как текст

Автоматически созданное описание

* + - 1. Выполните команду jobs и проверьте, что задание [1] остановлено. Переключитесь в виртуальный терминал tty2 и визуально проверьте, что задание [1] остановлено.

Изображение выглядит как текст

Автоматически созданное описание

* + - 1. Возобновите выполнение задания [1], используя команду kill, которая посылает процессу сигнал SIGCONT (18). Используйте команду jobs и виртуальный терминал tty2 для проверки того, что все три задания опять работают.

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

* + - 1. Завершите работу всех трех процессов. Если вы не задаете сигнал, который нужно послать процессу, то команда kill посылает по умолчанию сигнал SIGTERM (15), который вызывает завершение процесса. После посылки сигналов заданиям [2] и [3], используйте команду jobs, чтобы проверить завершение работы этих заданий:

*$ kill %2 %3*

*$ jobs*

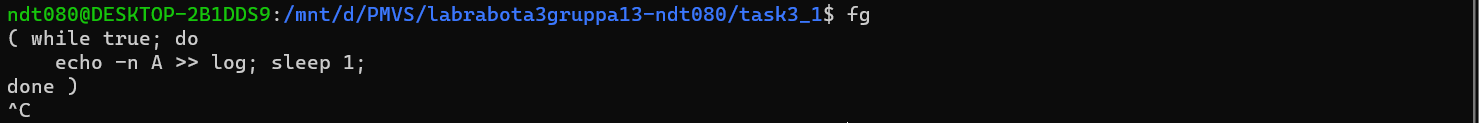
Изображение выглядит как текст

Автоматически созданное описание

* + - 1. Чтобы завершить работу последнего процесса, выполните команды:

*$ fg*

*$ <Ctrl+c>*



* + - 1. Выполните команду jobs и проверьте, что больше заданий не выполняется. Переключитесь в виртуальный терминал tty2 и визуально проверьте, что файл ~/log не увеличивается. Остановите исполнение команды tail, нажав клавиши <Ctrl+c>, и завершите сеанс на виртуальном терминале tty2.



Изображение выглядит как текст

Автоматически созданное описание

* + - 1. Переключитесь в виртуальный терминал tty1 и удалите файл ~/log.

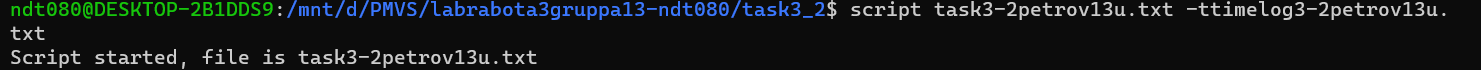
Изображение выглядит как текст, экран, снимок экрана

Автоматически созданное описание

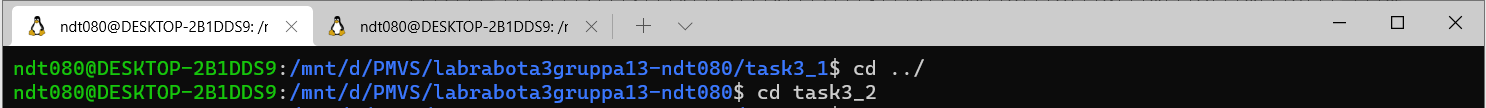
## ЗАДАНИЕ 3.2. УПРАВЛЕНИЕ ПРОЦЕССАМИ

Изучите примеры задания 3.2 и выполните их в ОС Ubuntu.

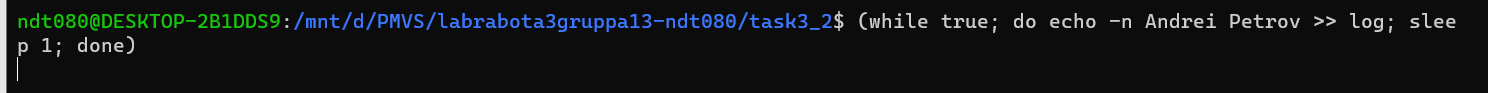
Включите ведение протокола командой script с журналом меток времени. **Протокол** назвать по следующему шаблону — taskXФамилияNM, где X — номер выполняемого задания, Фамилия — заменить на вашу фамилию латиницей и строчными буквами, N — номер группы, например 12 или 13, M — **u** — для Ubuntu. **Журнал меток** назвать по следующему шаблону — timelogXФамилияNM, где X — номер выполняемого задания, Фамилия — заменить на вашу фамилию латиницей и строчными буквами, N — номер группы, например 12 или 13, M — **u** — для Ubuntu.



1. Войдите в систему на виртуальных терминалах 1 и 2 (tty1, tty2) под учетной своей записью.



1. Переключитесь в терминал tty1 и запустите на переднем плане бесконечный процесс.

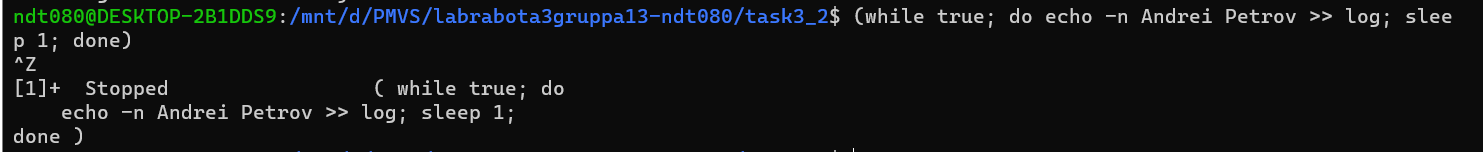


1. Переключитесь в виртуальный терминал tty2 и проверьте работу процесса, запущенного на пункте 2.

Изображение выглядит как текст

Автоматически созданное описание

1. Переключитесь в виртуальный терминал tty1 и приостановите работающий процесс.

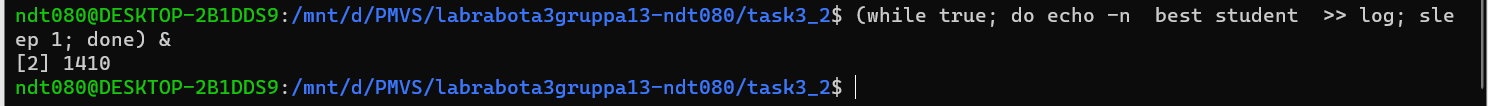


1. Возобновите работу процесса в виртуальном терминале tty1 в фоновом режиме.

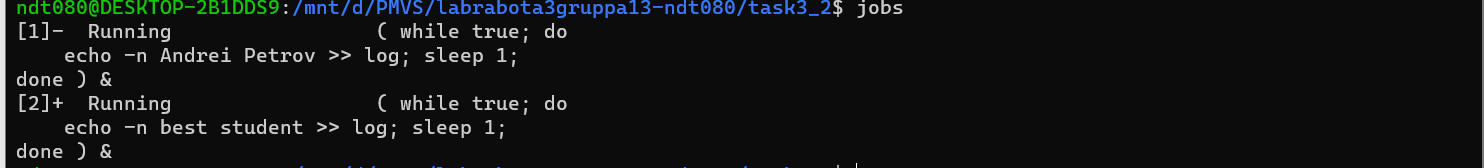
Изображение выглядит как текст

Автоматически созданное описание

1. Запустите в виртуальном терминале tty1 второй бесконечный процессов запущенных в фоновом режиме.



1. Выполните команду jobs и проверьте, что два запущенных процесса работают.



1. Установите приоритет процесса, запущенного в пункте 2, равным 10.

Изображение выглядит как текст

Автоматически созданное описание

1. Остановите процесс, запущенный в пункте 6, командой kill.

Изображение выглядит как текст

Автоматически созданное описание

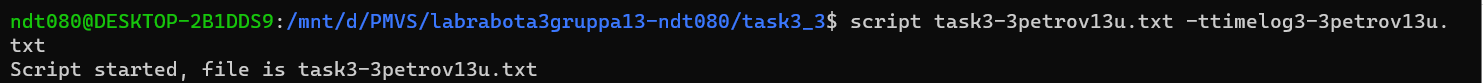
1. Проверьте, что работает только один процесс.
2. Прервите исполнение работающего процесса и проверьте, что не осталось работающих процессов.



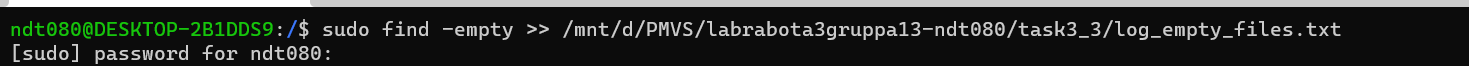
## ЗАДАНИЕ 3.3. УПРАВЛЕНИЕ ПРОЦЕССАМИ И ПЛАНИРОВА- НИЕ ЗАДАНИЙ

Изучите примеры задания 3.3 и выполните их в ОС Ubuntu.

Включите ведение протокола командой script с журналом меток времени. **Протокол** назвать по следующему шаблону — taskXФамилияNM, где X — номер выполняемого задания, Фамилия — заменить на вашу фамилию латиницей и строчными буквами, N — номер группы, например 12 или 13, M — **u** — для Ubuntu. **Журнал меток** назвать по следующему шаблону — timelogXФамилияNM, где X — номер выполняемого задания, Фамилия — заменить на вашу фамилию латиницей и строчными буквами, N — номер группы, например 12 или 13, M — **u** — для Ubuntu.



1. Найдите файлы а) пустые б) скрытые в домашнем каталоге в фоновом режиме и результат сохраните в файл со своей фамилией.





1. Запустите в фоновом режиме два задания: sleep 200 и sleep 2500, выведите информацию о состоянии заданий. Снимите с выполнения второе задание, выведите информацию о заданиях.

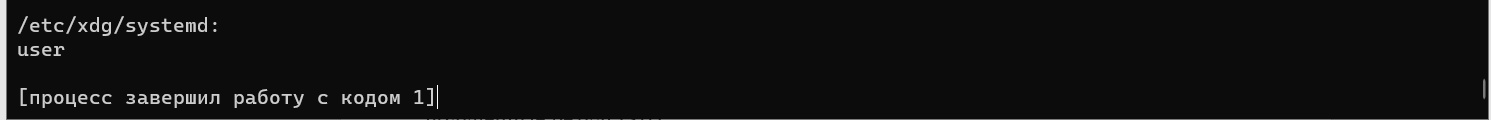
Изображение выглядит как текст

Автоматически созданное описание

1. Выполните команду exec ls -R /etc. Изучите её поведение.

Изображение выглядит как текст

Автоматически созданное описание



*exec* – это команда с двумя очень разными вариантами поведения, в зависимости от того, используется ли с ней хотя бы один аргумент или вообще не используется аргумент.

Если команда, переданная в качестве первого аргумента, не существует, текущая оболочка, а не только *exec* команда, завершается с ошибкой, если оболочка не является интерактивной или не установлена ​​опция *bash execfail*

Если команда существует и является исполняемой, она заменяет текущую оболочку.

Запуск сценария из командной строки приводит к запуску новой оболочки, которая и будет выполнять список команд, содержащихся в файле сценария. Другими словами, любой сценарий (или программа) запускается как дочерний процесс родительской командной оболочки. Однако, программа, выполняемая по команде exec, заменяет текущую программу, и поэтому в системе остается на один выполняемый процесс меньше.

Так мы запустили рекурсивный вывод списка файлов и директорий, находящихся в /etc, которая заменила текущий процесс терминала.

1. Запустите порожденную оболочку bash. Исследуйте, посылая родительской оболочке сигналы TERM, INT, QUIT и HUP, что при этом происходит?

*TERM* - игнорируется *INT* - очищает командную строку

*HUP* - завершает сеанс

1. От имени обычного пользователя пошлите сигнал KILL любому процессу, запущенному от имени другого пользователя. Что произойдет?

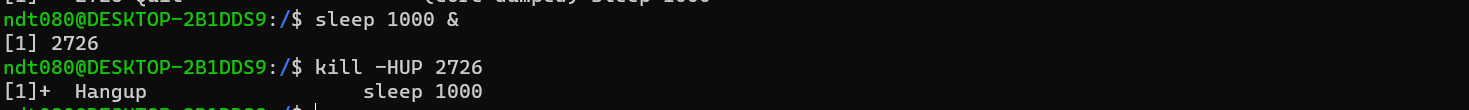
Ничего не произойдет.

1. Запустите в фоновом режиме команду sleep 1000. Проверьте, на какие сигналы из следующих: TERM, INT, QUIT и HUP, реагирует эта команда.









1. Запрограммируйте оболочку так, чтобы при получении ей сигнала TERM создавался файл pwd.txt, содержащий информацию о текущем каталоге и текущем пользователе.

Изображение выглядит как текст

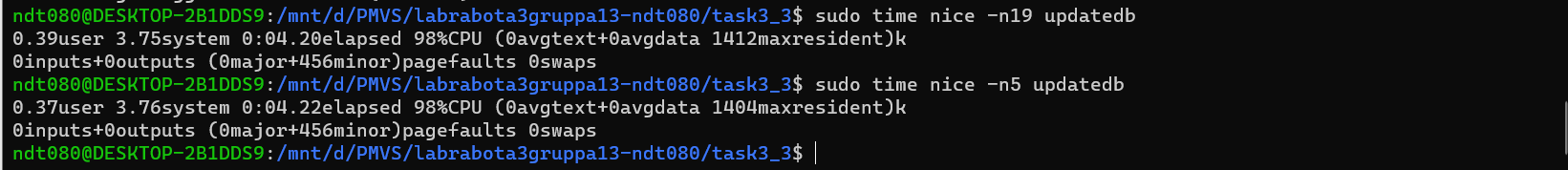
Автоматически созданное описание

1. Запустите порожденную оболочку. Работает ли в ней созданный обработчик?
2. От имени обычного пользователя попытайтесь запустить оболочку bash со значением nice number, равным 1. Какое сообщение выводится?

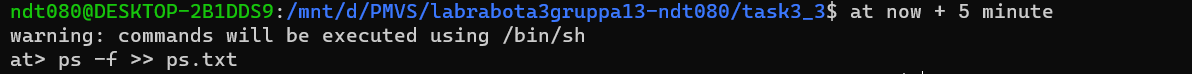


1. От имени суперпользователя запустите команду индексирования базы данных поиска вследующем виде: time nice -n19 updatedb. Затем выполните такую же команду, в которой значение nice number для updatedb будет 5. Сравните полученные результаты.

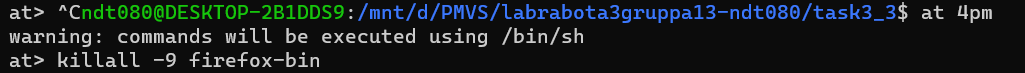




1. При помощи команды at сделать так, чтобы ровно через 5 минут от текущего времени произошла запись списка всех процессов в файл с именем, содержащим в своём названии системное время на момент записи.



1. При помощи команды at организовать обычное завершение работы браузера firefox или chrome в 16:00.



1. Сделать при помощи cron так, чтобы команда updatedb запускалась раз в сутки, каждый час, каждые 5 минут.

Изображение выглядит как текст

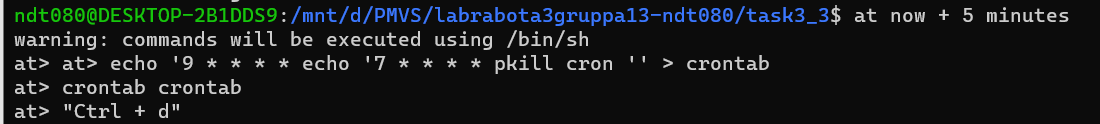
Автоматически созданное описание

1. При помощи cron организовать убийство браузера firefox и chrome каждые 10 минут.

Изображение выглядит как текст

Автоматически созданное описание

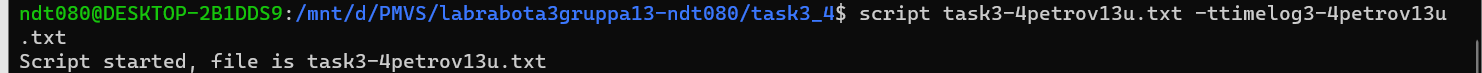
1. При помощи команды at сделать так, чтобы через 5 минут от текущего времени создалось задание для cron, которое создавало бы каждые 9 минут ещё одно задание для cron, заключающееся в том, чтобы каждые 7 минут уничтожать все задания пользователя для cron.



## Задание 3.4. Управление процессами и планирование заданий

Изучите задачи задания 3.4 и выполните их в ОС Raspberry PI.

Выполните задачи ниже и результат запуска скрипта из каждой задачи залогируйте с помощью команды script с журналом меток времени. **Протокол** назвать по следующему шаблону — taskXФамилияNM, где X — номер выполняемого задания, Фамилия — заменить на вашу фамилию латиницей и строчными буквами, N — номер группы, например 12 или 13, M — **r** — для Raspberry PI. **Журнал меток** назвать по следующему шаблону — timelogXФамилияNM, где X — номер выполняемого задания, Фамилия — заменить на вашу фамилию латиницей и строчными буквами, N — номер группы, например 12 или 13, M — **r** — для Raspberry PI.



**Задача 1.** Создайте и запустите следующий сценарий, который представляет бесконечный процесс, выводящий значение счетчика каждую секунду и завершающий свою работу при нажатии клавиш <Ctrl+c>.

#!/bin/bash # trap test

trap ’echo you hit Ctrl+c; exit’ SIGINT count=0

while:

do

sleep 1

count=$(expr $count + 1) echo $count

done

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

**Задача 2.** Написать пример программы, которая запускает и связывает каналом два процесса: вывод содержимого каталога и подсчет количества строк (ls и wc).

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

**Задача 3.** Изучить параграф **10.10. «Функции alarm и pause»** из книги [У.](https://edufpmi.bsu.by/mod/url/view.php?id=902) [Ричард Стивенс, Стивен А. Раго. UNIX. Профессиональное программирование. 3-е](https://edufpmi.bsu.by/mod/url/view.php?id=902) [изд](https://edufpmi.bsu.by/mod/url/view.php?id=902). Написать пользовательскую функцию обработки сигнала. Установка обработки сигнала происходит одноразово (обрабатывается только одно событие, связанное с появлением данного сигнала SIG\_ALRM). Возврат из функции-обработчика происходит в точку прерывания процесса.

Изображение выглядит как текст

Автоматически созданное описание

## Задание 3.5. Управление процессами

Перед началом выполнения задания изучите материалы глав 8-12 из книги [У.](https://edufpmi.bsu.by/mod/url/view.php?id=902) [Ричард Стивенс, Стивен А. Раго. UNIX. Профессиональное программирование. 3-е](https://edufpmi.bsu.by/mod/url/view.php?id=902) [изд](https://edufpmi.bsu.by/mod/url/view.php?id=902).

Результаты тестирования работы скриптов залогируйте, т. е. включите ведение протокола командой script с журналом меток времени. **Протокол** назвать по следующему шаблону — taskXФамилияNM, где X — номер выполняемого задания, Фамилия — заменить на вашу фамилию латиницей и строчными буквами, N — номер группы, например 12 или 13, M — **u** — для Ubuntu. **Журнал меток** назвать по следующему шаблону — timelogXФамилияNM, где X — номер выполняемого задания, Фамилия — заменить на вашу фамилию латиницей и строчными буквами, N — номер группы, например 12 или 13, M — **u** — для Ubuntu.

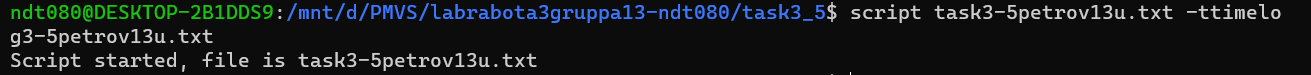
### Варианты индивидуальных заданий

Номер индивидуального варианта K равен числу букв вашей фамилии (N1), умноженному на число букв по паспорту вашего имени (N2), умноженному на число букв Вашего отчества (N3) по модулю 23:

K= (N1\*N2\*N3) mod 23

В каждой программе должен быть контроль ошибок для всех операций с файлами и каталогами.

**K = (6\*6\*13) mod 23 = 8**



Написать программу поиска заданной пользователем комбинации из m байт (m < 255) во всех файлах текущего каталога. Пользователь задаёт в качестве аргументов командной строки имя каталога, строку поиска, файл результата. Главный процесс открывает каталог и запускает для каждого файла каталога отдельный процесс поиска заданной комбинации из m байт. Каждый процесс выводит на экран и в файл результата свой pid, полный путь и имя файла, число просмотренных в данном файле байт и результаты поиска (всё в одной строке!). Результаты поиска (только найденные файлы) по предыдущему формату записываются в выходной файл. Число запущенных процессов в любой момент времени не должно превышать N (вводится пользователем). Проверить работу программы для каталога /usr/include/ и строки “stdio.h”

**Код программы:**

#include <stdio.h>  
#include <libgen.h>  
#include <fcntl.h>  
#include <string.h>  
#include <errno.h>  
#include <unistd.h>  
#include <pthread.h>  
#include <dirent.h>  
#include <sys/stat.h>  
#include <stdlib.h>  
#include <syscall.h>  
  
#define ARGS\_COUNT 4  
  
typedef enum {  
 ST\_NULL,  
 ST\_BUSY,  
 ST\_FREE  
} thread\_status\_t;  
  
typedef struct {  
 thread\_status\_t \*thread\_status;  
 char \*filename;  
} thread\_params\_t;  
  
typedef struct {  
 int entries\_count;  
 ssize\_t total\_bytes;  
} find\_result\_t;  
  
char \*MODULE\_NAME;  
char \*BYTES\_SEQUENCE;  
pthread\_t \*THREADS;  
thread\_status\_t \*THREADS\_STATUS;  
  
int find\_bytes\_in\_file(const char \*filename, const char \*bytes\_sequence, find\_result\_t \*find\_result);  
  
int wait\_for\_thread(int threads\_count);  
  
void print\_error(const char \*module\_name, const char \*error\_msg, const char \*file\_name) {  
 fprintf(stderr, "**%s**: **%ld**: **%s %s\n**", module\_name, syscall(SYS\_gettid), error\_msg, file\_name ? file\_name : "");  
}  
  
void print\_result(const char \*filename, const find\_result\_t find\_result) {  
 printf("**%ld %s %ld %d\n**", syscall(SYS\_gettid), filename, find\_result.total\_bytes, find\_result.entries\_count);  
}  
  
void \*worker(void \*args) {  
 thread\_params\_t \*thread\_params = (thread\_params\_t \*) args;  
 find\_result\_t find\_result;  
  
 if (find\_bytes\_in\_file(thread\_params->filename, BYTES\_SEQUENCE, &find\_result) != -1) {  
 if (find\_result.entries\_count != 0) {  
 print\_result(thread\_params->filename, find\_result);  
 }  
 };  
  
 \*(thread\_params->thread\_status) = ST\_FREE;  
  
 while (\*(thread\_params->thread\_status) != ST\_NULL);  
  
 free(thread\_params->filename);  
 free(thread\_params);  
  
 return **NULL**;  
}  
  
int find\_bytes\_in\_file(const char \*filename, const char \*bytes\_sequence, find\_result\_t \*find\_result) {  
 int fd = open(filename, O\_RDONLY);  
 if (fd == -1) {  
 print\_error(MODULE\_NAME, strerror(errno), filename);  
 return -1;  
 }  
  
 int i = 0;  
 int entries\_count = 0;  
 ssize\_t total\_bytes = 0;  
 size\_t sequence\_len = strlen(bytes\_sequence);  
 char c = 0;  
  
 ssize\_t bytes\_read;  
 while ((bytes\_read = read(fd, &c, sizeof(char))) > 0) {  
 total\_bytes++;  
 (bytes\_sequence[i] == c) ? (i++) : (i = 0);  
 if (sequence\_len == i) {  
 entries\_count++;  
 i = 0;  
 }  
 }  
  
 if (bytes\_read == -1) {  
 print\_error(MODULE\_NAME, strerror(errno), filename);  
 return -1;  
 }  
  
 if (close(fd) == -1) {  
 print\_error(MODULE\_NAME, strerror(errno), filename);  
 return -1;  
 }  
  
 find\_result->entries\_count = entries\_count;  
 find\_result->total\_bytes = total\_bytes;  
  
 return 0;  
}  
  
void file\_path(char \*dest, const char \*const path, const char \*const name) {  
 strcpy(dest, path);  
 strcat(dest, "/");  
 strcat(dest, name);  
}  
  
void find\_bytes\_in\_folder(const char \*path, const int threads\_count) {  
 DIR \*dir\_stream;  
 struct dirent \*dir\_entry;  
 if (!(dir\_stream = opendir(path))) {  
 print\_error(MODULE\_NAME, strerror(errno), path);  
 return;  
 }  
  
 thread\_params\_t \*thread\_params;  
 pthread\_t thread;  
  
 errno = 0;  
 while ((dir\_entry = readdir(dir\_stream)) != **NULL**) {  
 char \*entry\_name = dir\_entry->d\_name;  
  
 if (!strcmp(".", entry\_name) || !strcmp("..", entry\_name))  
 continue;  
  
 char \*full\_path = malloc(strlen(entry\_name) + strlen(path) + 2);  
 file\_path(full\_path, path, entry\_name);  
 struct stat entry\_info;  
 if (lstat(full\_path, &entry\_info) == -1) {  
 print\_error(MODULE\_NAME, strerror(errno), full\_path);  
 errno = 0;  
 continue;  
 }  
  
 if (S\_ISDIR(entry\_info.st\_mode)) {  
 find\_bytes\_in\_folder(full\_path, threads\_count);  
 } else {  
 if (S\_ISREG(entry\_info.st\_mode)) {  
 int thread\_id = wait\_for\_thread(threads\_count);  
 THREADS\_STATUS[thread\_id] = ST\_NULL;  
 if (pthread\_join(THREADS[thread\_id], **NULL**) == -1){  
 print\_error(MODULE\_NAME, strerror(errno), **NULL**);  
 return;  
 };  
  
 thread\_params = malloc(sizeof(thread\_params\_t));  
 thread\_params->thread\_status = &(THREADS\_STATUS[thread\_id]);  
 thread\_params->filename = full\_path;  
  
 THREADS\_STATUS[thread\_id] = ST\_BUSY;  
 if (pthread\_create(&thread, **NULL**, &worker, thread\_params) == -1) {  
 print\_error(MODULE\_NAME, strerror(errno), **NULL**);  
 return;  
 };  
  
 THREADS[thread\_id] = thread;  
 }  
 }  
 }  
 if (errno) {  
 print\_error(MODULE\_NAME, strerror(errno), path);  
 }  
  
 if (closedir(dir\_stream) == -1) {  
 print\_error(MODULE\_NAME, strerror(errno), path);  
 }  
  
}  
  
int wait\_for\_thread(int threads\_count) {  
 int i = 0;  
 while (THREADS\_STATUS[i] == ST\_BUSY) {  
 i++;  
 if (i == threads\_count) {  
 i = 0;  
 }  
 }  
 return i;  
}  
  
char all\_finished(int threads\_count) {  
 int i;  
 for (i = 0; (i < threads\_count); i++) {  
 if (THREADS\_STATUS[i] != ST\_FREE) {  
 return 0;  
 }  
 }  
 return 1;  
}  
  
int main(int argc, char \*argv[]) {  
 int threads\_count;  
 MODULE\_NAME = basename(argv[0]);  
  
 if (argc != ARGS\_COUNT) {  
 print\_error(MODULE\_NAME, "Wrong number of parameters", **NULL**);  
 return 1;  
 }  
  
 if ((threads\_count = atoi(argv[3])) < 1) {  
 print\_error(MODULE\_NAME, "Thread number must be bigger than 1.", **NULL**);  
 return 1;  
 }  
  
 THREADS\_STATUS = calloc(sizeof(thread\_status\_t), (size\_t) threads\_count);  
 for (int i = 0; i < threads\_count; i++) {  
 THREADS\_STATUS[i] = ST\_NULL;  
 }  
 BYTES\_SEQUENCE = argv[2];  
 THREADS = calloc(sizeof(pthread\_t), (size\_t) threads\_count);  
  
 find\_bytes\_in\_folder(argv[1], threads\_count);  
 while (!all\_finished(threads\_count));  
}

**Собираем программу:**



**Выполняем программу:**

Параметры:

- путь /etc

- комбинация system

- количество потоков 2

- стандартный вывод перенаправляем в файл task\_result.txt



Изображение выглядит как текст

Автоматически созданное описание

## Задание 3.6. Управление процессами: Творческое задание для выполнения в группе

Сформировать группу из 2-х человек и выполнить задание с реализацией подключения по ssh и запуска. Для автоматического входа на удаленный компьютер сгенерировать и использовать ключи. Необходимо разбиться на пары (драконвоин) сначала опробовать модельное сражение (вручную вводя команды), затем реализовать скрипты.

Необходимо создать скрипт воина (warrior.sh), который запрашивает ip-адрес компьютера замка (castle), заходит на него по протоколу ssh, и должен забрать с этого компьютера на свой при помощи команды scp похищенный герб (gerb.gif). Воину противостоит дракон (dragon.sh), который в начале работы прячет герб (gerb.gif) в случайное место файловой системы своего компьютера, а далее осуществляет охрану замка: с некоторой периодичностью просматривает список процессов и, если обнаруживает воина, то убивает его (убивать службу sshd запрещается).

Сражаться разрешается при помощи сигналов 2 (INT) и 15 (TERM).

Защищаться можно при помощи ловушек (trap).

Воин также может (и должен) убить дракона.

Если останется время, сразиться с противниками из другой группы.

#### Задание засчитывается при предоставлении результатов модельного сражения (ручной ввод команд), результатов выполнения скриптов и результатов сражения с другой командой.

# Контрольные вопросы

1. Объясните понятия процесса и ресурса. Какое их значение в организации вычислительного процесса в ОС Linux?

Процесс – набор выполняемых заданий, инструкций.

Ресурс – это то, что выделяется процессу для выполнения данных задач.

Процесс представлен в памяти в виде дескриптора процесса, который включает: исполняемый текст программы и управляющая информация. Концепция процессов обеспечивает свойство многозадачности ОС, т.е. когда в активном состоянии могут находиться несколько задач, между которыми разделяется процессорное время.

1. Какая информация содержится в описателях процессов? Как просмотреть их содержание в процессе работы с системой?

Описатель процесса – это дескриптор. Дескриптор включает в себя исполняемый текст программы задачи, управляющую информацию (контекст процесса), необходимую для выделения и использования выделенных процессу ресурсов: приоритет, необходимое адресное пространство, вектор прерывания (слово состояния программы), величину кванта процессорного времени, признаки состояния процесса и пр.

Посмотреть: ps -l, ps.

1. Какими способами можно организовать выполнение программ в фоновом режиме?

Для запуска фонового (параллельного с другими потомками) процесса в командную строку необходимо и достаточно последним символом добавить знак &

1. Какие особенности выполнения программ в фоновом режиме? Как избежать вывода фоновых сообщений на экран и прерывания выполнения фоновых программ при прекращении сеанса работы с системой?

Фоновые процессы обладают некоторыми недостатками:

1. не допускают ввода с клавиатуры;
2. обеспечивают вывод на экран, но при этом нарушают целостность вывода диалогового процесса.

Общепринятый прием исключения влияния фонового вывода на интерактивную работу:

$<командная\_строка> > имя\_файла.out &

* <командная\_строка> планирует задание для фонового режима;
* перенаправляет вывод вместо экрана в указанный файл головного каталога пользователя.

Oсобенности работы с фоновым режимом:

* выполняемая в фоновом режиме программа (команда), требующая стандартного ввода, должна читать его из файла с использованием перенаправленного ввода;
* программа, выполняемая в фоновом режиме, не может быть прервана <Ctrl\*C>, так как она отсоединяется от клавиатуры и может быть прекращена только с помощью команды kill или выходом из системы;
* выход из системы exit надо выполнять два раза: для завершения фонового процесса и завершения основного процесса shell.

1. Как пользователь может повлиять на распределение ресурсов между активными процессами?

С помощью команды nice.

Если при выполнении задания образуются несколько порожденных процессов, то все они имеют одинаковый приоритет равный родительскому. В этом случае все процессы получают ресурсы равными долями (простой режим разделения времени). При необходимости выделения наиболее важных родительских процессов порожденным второстепенным можно понизить приоритет с помощью команды.

1. Как можно прервать выполнение активных процессов? Какая информация для этого необходима и откуда она извлекается?

Завершение процессов – одна из функций управления процессами.

Прекратить выполнение любого процесса можно с помощью команды:

kill [-опции] PID1 [PID2......] — передает сигнал процессу PID.

1. Перечислите базовые средства взаимодействия процессов в Linux.

Каналы, сигналы, разделяемая память, очереди сообщений, сокеты

1. Поясните особенности работы с каналами в Linux.

Канал – это средство связи стандартного вывода одного процесса со стандартным вводом другого. Эта особенность широко используется даже в командной строке UNIX (в shell-е). Когда процесс создает канал, ядро устанавливает два файловых дескриптора для пользования этим каналом. Один такой дескриптор используется, чтобы открыть путь ввода в канал (запись), в то время как другой применяется для получения данных из канала (чтение)

1. Почему отложенные вызовы не обрабатываются непосредственно обработчиком прерывания таймера?

Так как выполнении функции отложенных вызовов происходит в системном контексте, а не в контексте прерывания.