МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Тихоокеанский государственный университет»

Кафедра «Программное обеспечение вычислительной техники и автоматизированных систем»

Лабораторная работа № 2

По предмету «Операционные системы и системное ПО»

«ПРОЦЕССЫ И ПОТОКИ В ОПЕРАЦИОННОЙ СИСТЕМЕ LINUX»

Выполнил:

студент группы ПО(аб) –71

Маслеников М.В.

Проверил :

Тормозов В.С.

Хабаровск – 2020г.

Цель работы: изучение принципов создания, работы и завершения процессов и потоков в операционной системе Linux; изучение функций библиотеки Pthreads.

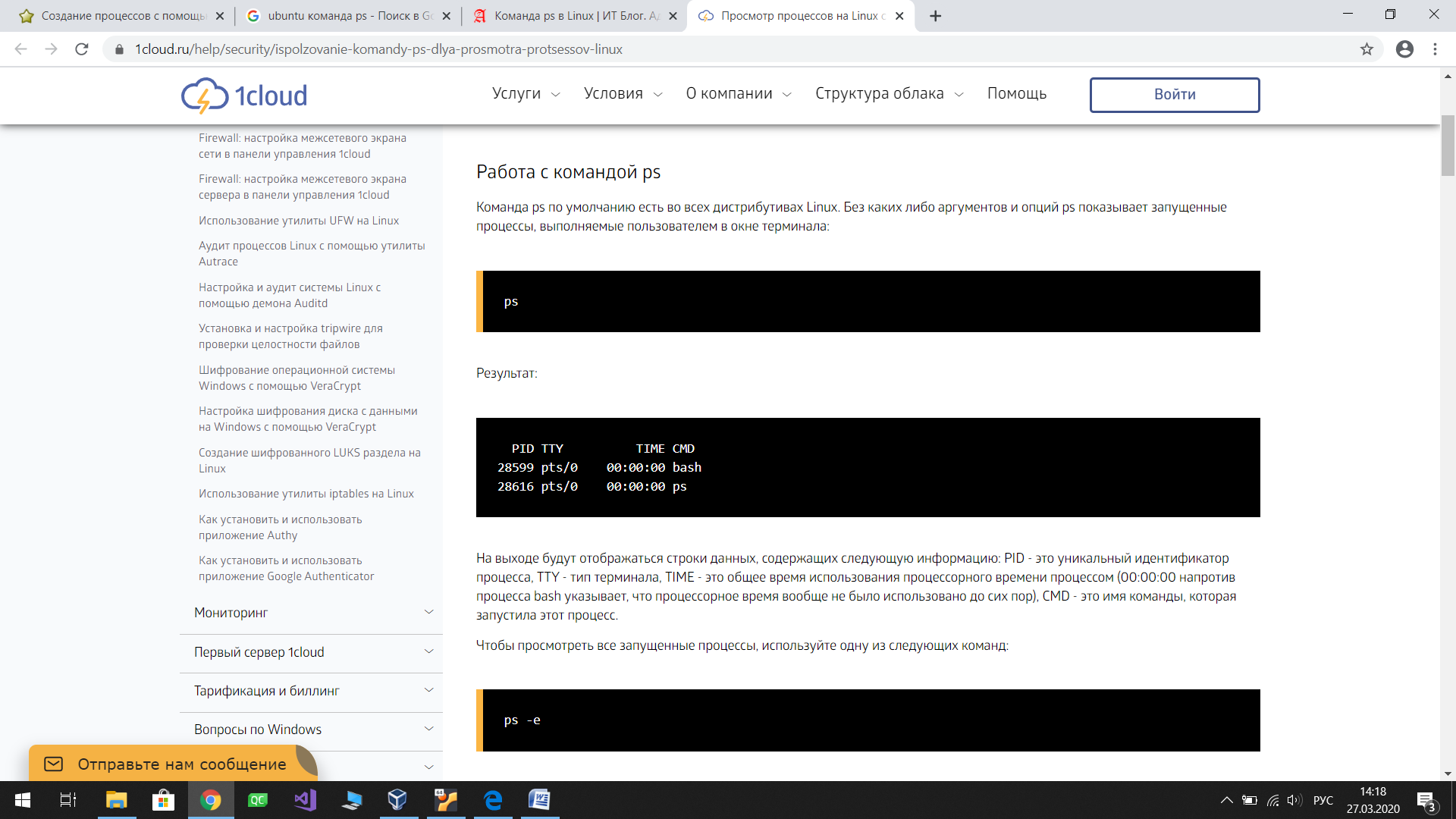
**Порядок выполнения работы**

**Часть 1 – Процессы.**

1. **Команды ps и kill**
   1. **Команда ps**

Команда **ps** по умолчанию есть во всех дистрибутивах Linux. Без каких либо аргументов и опций ps показывает запущенные процессы, выполняемые пользователем в окне терминала.

Результат:



На выходе будут отображаться строки данных, содержащих следующую информацию: **PID** - это уникальный идентификатор процесса, **TTY** - тип терминала, **TIME** - это общее время использования процессорного времени процессом (00:00:00 напротив процесса bash указывает, что процессорное время вообще не было использовано до сих пор), **CMD** - это имя команды, которая запустила этот процесс.

Чтобы просмотреть все запущенные процессы, используется команда **ps –e**.

Чтобы просмотреть все процессы, связанные с этим терминалом, используется команда **ps T**.

Чтобы просмотреть все работающие(running) процессы, используется команда **ps r**.

Можно вывести определенные процессы с помощью ps с помощью различных критериев выбора.

Например, если известен идентификатор процесса PID, можно использовать следующую команду: **ps –p <pid>**

Вы можете выбрать несколько процессов, указав несколько идентификаторов процесса следующим образом**: ps –p “<pid1> <pid2> …”**

Процесс можно искать и по команде: **ps –C <команда>.**

Для получения информации обо всех доступных опциях ps используется команда **man ps** в терминале.

* 1. **Команда kill**

Один из способов «убить» запущенное приложение в Linux - это использование такой команды, как **kill.**

Для этого нам необходимо узнать PID процесса, который нужно «убить». Можно использовать команду ps, описанную в пункте 1.1.

Еще один способ узнать PID процесса — это **команда pidof**, которая принимает в качестве параметра название процесса и выводит его PID: **pidof <название процесса>.**

Когда стал известен PID процесса, мы можем убить его **командой kill**. Команда kill принимает в качестве параметра PID процесса. Например, убьем процесс с номером 25609: **kill 25609**

Вообще команда kill предназначена для посылки сигнала процессу. По умолчанию, если мы не указываем какой сигнал посылать, посылается сигнал SIGTERM (от слова termination — завершение). SIGTERM указывает процессу на то, что необходимо завершиться. Каждый сигнал имеет свой номер. SIGTERM имеет номер 15. Список всех сигналов (и их номеров), которые может послать команда kill, можно вывести, выполнив kill -l. Чтобы послать сигнал SIGKILL (он имеет номер 9) процессу 25609, выполните в командой строке: **kill -9 25609**

Сигнал SIGTERM может и не остановить процесс (например, при перехвате или блокировке сигнала), SIGKILL же выполняет уничтожение процесса всегда, так как его нельзя перехватить или проигнорировать.

1. **Команды jobs, fg, bg, top.**
   1. Родительский процесс может либо ждать завершения дочернего, либо продолжать свое выполнение. Если в роли родителя выступает командная оболочка, то это значит, что процессы, запущенные с одной консоли (или виртуального терминала), распадаются на две группы: те, которых оболочка ждет, — они взаимодействуют с пользователем, занимая консоль, — и те, после запуска которых с пользователем взаимодействует сама оболочка (консоль свободна). Эти группы называются передним и задним планами.

Чтобы запустить процесс на заднем плане (в фоновом или асинхронном режиме), нужно завершить командную строку управляющим оператором &.

Подкоманды оболочки **jobs**, **bg** (background) и **fg** (foreground) позволяют манипулировать заданиями, выполняющимися на переднем и заднем планах:

Команда **jobs** выводит список процессов, которые выполняются в фоновом режиме,

**fg** <номер\_задания> переводит процесс на передний план,

**bg** <номер\_задания> — переводит процесс на задний план.

Номер задания — это не PID, а число, которое команда **jobs** выводит в квадратных скобках. С ключом **-I** она будет выводить, кроме того, и PID процесса.

Поскольку перечисленные команды — не самостоятельные утилиты, а подкоманды **bash**, справку по ним нужно запрашивать так: help <команда>.

* 1. Любой сервер, каким бы мощным он ни был, имеет ограниченный объем ресурсов. Каждая программа, работающая в активном или фоновом режиме, использует определенное количество виртуальной и физической памяти, процессорного времени и т.д. Иными словами, создает определенную нагрузку на сервер. Чтобы посмотреть, насколько система загружена в данный момент времени, используют консольную команду **top**.

Команда top в Linux системах позволяет вывести в виде таблицы перечень запущенных процессов и оценить, какой объем ресурсов они потребляют, т.е., какую нагрузку создают на сервер и дисковую подсистему. Такая информация помогает в дальнейшем оптимизировать работу системы.

1. **Функции execv(), execl(), execvp(), execlp().**

Функция **exec**() (execute) загружает и запускает другую программу. Таким образом, новая программа полностью замещает текущий процесс. Новая программа начинает свое выполнение с функции main. Все файлы вызывающей программы остаются открытыми. Они также являются доступными новой программе. Используется шесть различных вариантов функций exec.

#include <unistd.h>

int **execl**(char \*name, char \*arg0, ... /\*NULL\*/);

int **execv**(char \*name, char \*argv[]);

int execle(char \*name, char \*arg0, ... /\*,NULL,

char \*envp[]\*/);

int execve(char \*name, char \*arv[], char \*envp[]);

int **execlp**(char \*name, char \*arg0, ... /\*NULL\*/);

int **execvp**(char \*name, char \*argv[]);

Вызов exec происходит таким образом, что переданная в качестве аргумента программа загружается в память вместо старой, которая вызвала exec. Старой программе больше не доступны сегменты памяти, которые перезаписаны новой программой.

Суффиксы **l, v, p**, e в именах функций определяют формат и объем аргументов, а также каталоги, в которых нужно искать загружаемую программу:

**l** **(список)**. Аргументы командной строки передаются в форме списка arg0, arg1.... argn, NULL. Эту форму используют, если количество аргументов известно;

**v (vector)**. Аргументы командной строки передаются в форме вектора argv[]. Отдельные аргументы адресуются через argv [0], argv [1]... argv [n]. Последний аргумент (argv [n]) должен быть указателем NULL;

**p** **(path)**. Обозначенный по имени файл ищется не только в текущем каталоге, но и в каталогах, определенных переменной среды PATH;

e (среда). Функция ожидает список переменных среды в виде вектора (envp []) и не использует текущей среды.

1. **Написать программу по заданным условиям:**

1) Программа должна создать четыре процесса и ожидать окончания их выполнения.

2) Каждый процесс и основная программа должны выводить на экран сообщения о начале и завершении своей работы в следующем формате: PID PPID “сообщение” “время сообщения”

3) Для получения текущего времени используйте функцию ctime() из <time.h>.

4) Процессы должны использовать функции, перечисленные по порядку: execv(), execl(), execvp()и execlp() с любыми командами ОС Linux внутри. Вы можете также использовать любые свои программы.

5) Программа должна анализировать и сообщать о причинах завершения процессов, используя переменную errno. Все значения ошибок определены в заголовочном файле <sys/errno.h>

**Код программы:**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <stdlib.h>

#include <sys/errno.h>

#include <sys/wait.h>

#include <time.h>

#define TEST "./ex2"

#define TEST1 "./ex3"

int main(int argc, char \*argv[])

{

time\_t It;

It = time(NULL);

printf("PID\_zad1 = %d , PPID\_zad1 = %d\n", getpid(), getppid());

printf("Программа %s будет выполнена...\n", argv[0]);

printf ("Time\_begin\_zad1 = %s", ctime(&It));

//printf("\nВыполняется %s", argv[0]);

int rc, cnt;

rc = fork();

if (rc == 0)

{

execv("ex", argv);

if (errno!=0)

perror ("\nProgram Error 1\_1: ");

//perror("execv failed");

exit(1);

}

wait(&rc);

rc = fork();

if (rc == 0)

{

execl("ex1", "ex1", "HELLO,", "WORLD", NULL);

if (errno!=0)

perror ("\nProgram Error 1\_2: ");

//perror("execl failed");

exit(2);

}

wait(&rc);

rc = fork();

if (rc == 0)

{

execvp(TEST, argv);

//perror("execvp failed");

if (errno!=0)

perror ("\nProgram Error 1\_3: ");

exit(3);

}

wait(&rc);

rc = fork();

if (rc == 0)

{

execlp(TEST1, TEST1, "hello", NULL);

if (errno!=0)

{

perror ("\nProgram Error 1\_4: ");

printf("err 1\_4 = %d\n", errno);

}

//perror("execlp failed");

exit(4);

}

wait(&rc);

for (int i = 0; i< 500000000; i++)

cnt++;

It = time(NULL);

printf("\nParent program ending\n");

printf ("Time\_end\_zad1 = %s", ctime(&It));

printf("err\_zad1 = %d\n", errno);

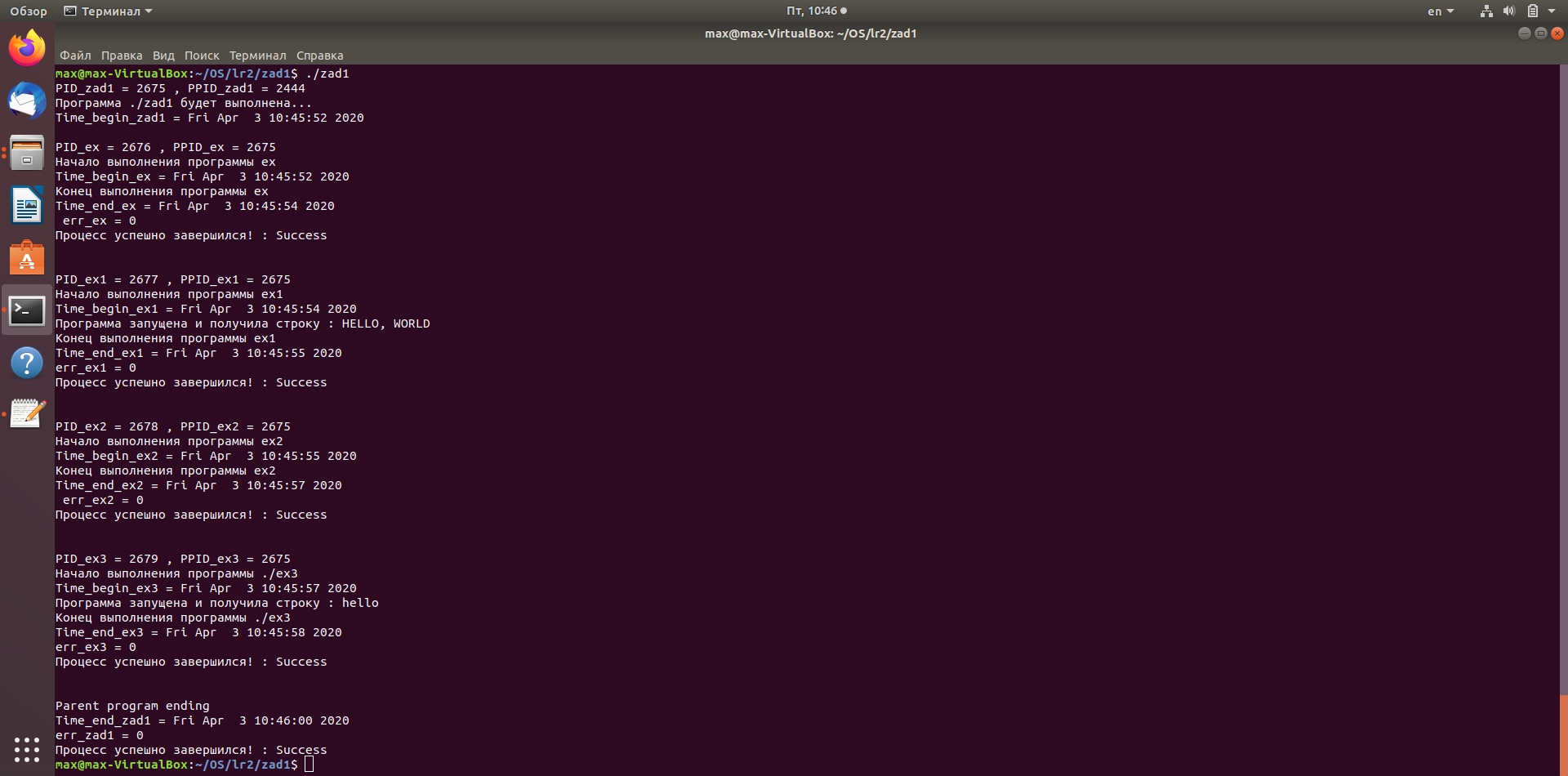
if (errno!=0)

perror ("\nProgram Error: ");

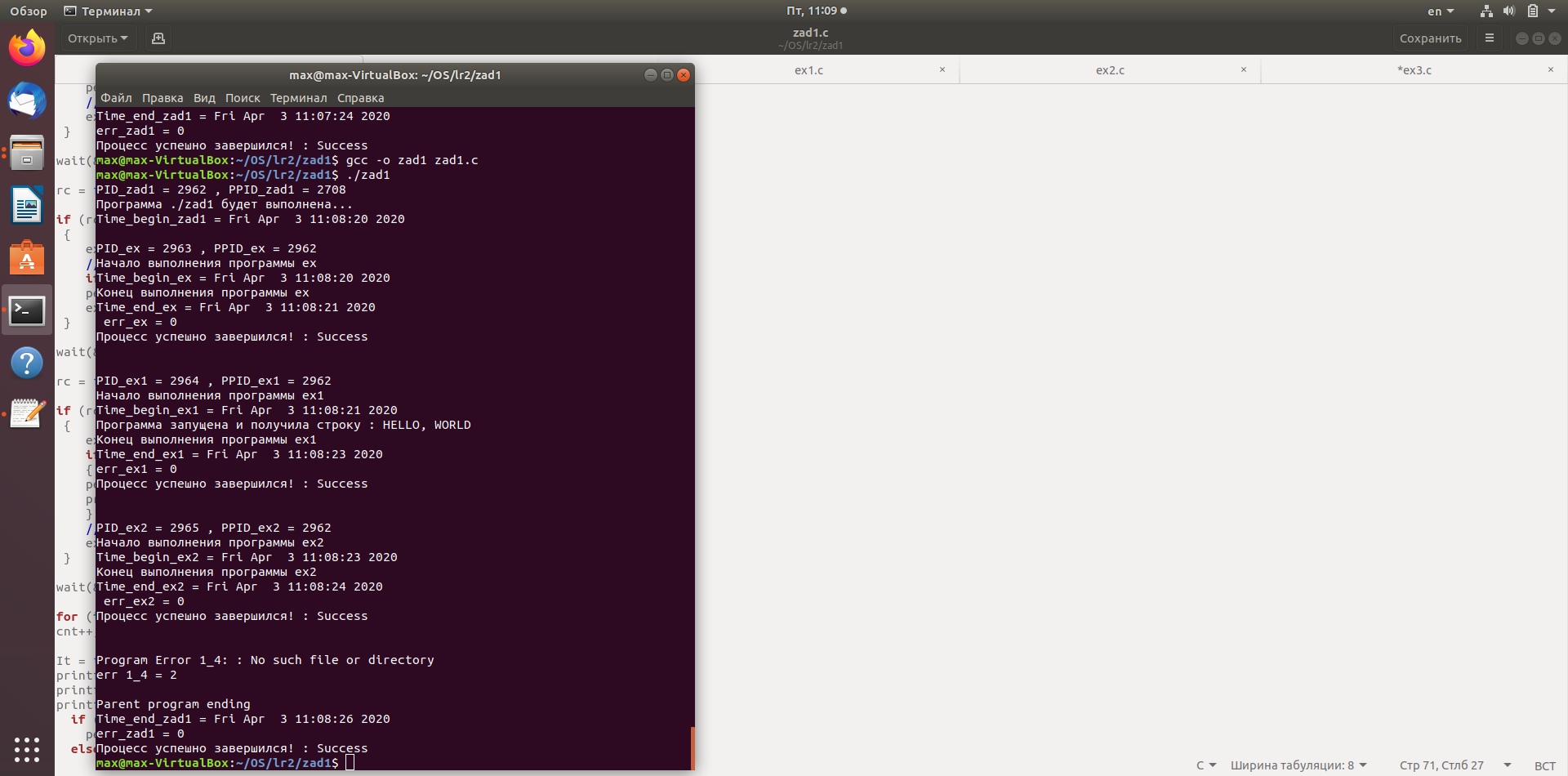
else perror("Процесс успешно завершился! "); return 0;}

**Результат работы программы:**

Каждый из процессов запускает отдельную программу, ниже представлен случай, когда все программы без ошибок, тогда функция perror( ) выводит значение переменной errno в строковом виде, в нашем случае – Success.



А в ниже представленном случае в программе, которую выполнял четвертый процесс, намеренно допущена ошибка, показывающая, что функция perror( ) не вернет значение “Success”, т.к. переменная errno = 2, а вернет значение “No such file or directory”:



**Часть 2 – Потоки.**

**2.1.** Запустить несколько раз программу, приведенную в примере 6. Объяснить:

А) Результат работы программы.

Б) Почему при запуске программы несколько раз и при увеличении числа потоков (NKIDS) получаются различные результаты?

**Код примера 6:**

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

int tot\_items = 0;

struct kidrec {

int data ;

pthread\_t id ; };

#define NKIDS 50

void \*kidfunc(void \*p)

{

int \*ip = (int \*)p;

int tmp, n;

tmp = tot\_items;

for (n = 0;n<50000;n++) tot\_items = tmp + \*ip;

}

int main ( )

{

struct kidrec kids[NKIDS];

int m;

for (m=0; m<NKIDS; ++m)

{

kids[m].data = m+1;

pthread\_create (&kids[m].id, NULL, kidfunc, &kids[m].data);

}

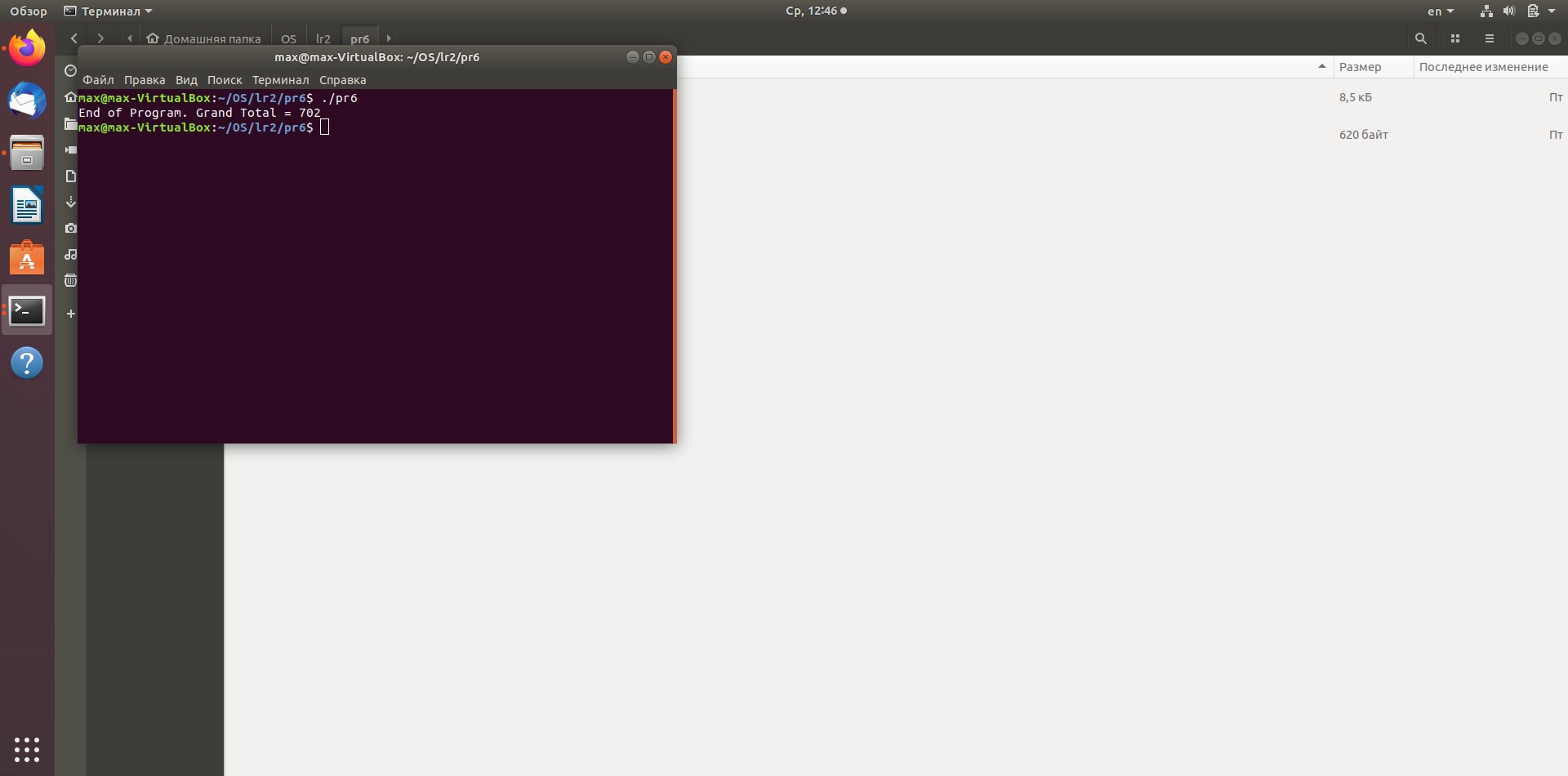
for (m=0; m<NKIDS; ++m) pthread\_join(kids[m].id, NULL);

printf ("End of Program. Grand Total = %d\n", tot\_items);

exit(0);

}

**Пример выполнения программы:**



**Объяснение работы программы:**

При создании потока выделяется отдельный участок памяти, запланированный под локальные переменные потока. В программе выполнение потоков производится параллельно, поэтому каждый поток при выполнении постоянно переписывает данный участок памяти в переменной tot\_items, чем объясняется непредсказуемость вывода этой переменной, и чем больше потоков, тем более непредсказуем будет результат, потому что будет неизвестно, какой из потоков перепишет переменную, так как они выполняется параллельно.

**2.2. Написать программу на языке C/C++, в которой:**

1) Запускается четное количество потоков параллельно.

2) Каждый нечетный поток (например, первый) создает файл с именем из своего PID, записывает в него произвольное число символов (от нескольких символов до сотен миллионов символов) и закрывает этот файл.

3) Каждый четный поток (например, второй) открывает файл, созданный предыдущим потоком с нечетным номером (в нашем случае первым), читает его, считает количество символов в файле и закрывает его; при этом четный поток не должен иметь никакой информации о количестве записываемых в файл символов и о том, закончена ли запись в файл нечетным потоком.

4) Каждый из потоков выводит следующую информацию: ThreadID, PID, PPID, время, имя файла, количество записанных или считанных символов.

5) Количество пар создаваемых потоков передается аргументом в программу с командной строки. Программа должна ждать завершения работы всех потоков, анализировать и сообщать о причинах завершения потоков. Для передачи имен файлом между потоками можно использовать символьный массив в основной программе.

**Код программы:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <sys/time.h>

#include <sys/types.h>

#include <unistd.h>

#include <pthread.h>

#include <dirent.h>

#include <string.h>

#include <sys/errno.h>

#include <sys/wait.h>

char\*\* files; // массив с именами файлов

void\* create\_file(void \*arg) // поток создания файла

{

int\* tmp =(int\*) arg; // индекс указывающий на имя файла

int ind = \*tmp;

int rec\_error;

free (tmp);

FILE\* fl;

long unsigned int PID;

int pid, ppid, num;

long unsigned int len, i = 0;

char buf;

time\_t vremya= time(NULL);

num = ind\*2 + 1; // номер потока

len = rand() %10000 + 3; // количество символов в файле

len +=2;

PID = pthread\_self(); // ThreadID

pid = getpid(); // PID

ppid = getppid(); // PPID

sprintf(files[ind], "%s/%lu%s", getenv("PWD"), PID, ".txt"); // имя файла

//fl = fopen(files[ind], "w");// создаем файл

if( fl = fopen(files[ind], "w")) // если файл создан

{

while( i!=len) // пока не запишем все символы

{

if( i % 2 ) buf = '0';

else buf = '1';

fputc(buf, fl);

i++;

}

if( fclose(fl) == EOF) printf("Fclose Error"); // закрываем файл

}

// выводим на экран информацию о потоке и файле

printf("\nWriting...\nThread #%i\n\tThreadID: %lu\n\tPID: %i\n\tPPID: %i\n\tTimeDate: %s\tFileName: %s\n\tLength: %lu\n\tErr: %d\n", num, PID, pid, ppid, ctime(&vremya), files[ind], len, errno);

pthread\_exit(NULL);

}

void\* read\_file(void \*arg) // поток читающий файл

{

int\* tmp =(int\*) arg; // индекс указывающий на имя файла

int ind = \*tmp;

free (tmp);

FILE\* fl;

long unsigned int PID;

int i = 0, pid, ppid, num;

long int len = 0;

//char t[32];

char buf[256];

time\_t vremya= time(NULL);

num = ind\*2 + 2; // номер потока

PID = pthread\_self(); // ThreadID

pid = getpid(); // PID

ppid = getppid(); // PPID

//fl = fopen(files[ind], "r"); // открываем файл для чтения

if( fl = fopen(files[ind], "r")) // если файл открылся

{

strcpy(buf, files[ind]); // считываем название файла

while( i!=EOF ) // пока не достигнем конца файла

{

i = fgetc(fl); // считываем посимвольно

if( i!=EOF ) len++; // счетчик символов

}

if( fclose(fl) == EOF) printf("Fclose Error"); // закрываем файл

}

//else // если файла нет

//{

// strcpy(buf, "file not found");

//}

// выводим результаты работы

printf("\nReading...\nThread #%i\n\tThreadID: %lu\n\tPID: %i\n\tPPID: %i\n\tTimeDate: %s\tFileName: %s\n\tLength: %lu\n\tErr: %d\n", num, PID, pid, ppid, ctime(&vremya), buf, len, errno);

pthread\_exit(NULL);

}

// запускаем приложение

int main(int argc, char\* argv[])

{

int i, qnt, res, tmp;

FILE\* fl;

pthread\_t\* ws;

pthread\_t\* rs;

srandom(time(NULL));

if( argc == 2 ) qnt = atoi(argv[argc-1]); // выбираем количество пар потоков из командной строки

else qnt = 0;

//if( qnt ) // если в командной строке прописано количество пар потоков

//{

ws = new pthread\_t[qnt]; // создаем указатели на потоки записи

rs = new pthread\_t[qnt]; // создаем указатели на потоки чтения

files = new char\*[qnt]; // создаем массив имен файлов

for(i=0; i<qnt;i++)

{

files[i] = new char[256]; // инициализируем строку имени файла

res = pthread\_create(&ws[i], NULL, create\_file, new int(i)); // запускаем нечетный поток

if (res != 0) // проверка на успех

{

perror("creation failed");

exit(EXIT\_FAILURE);

}

res = pthread\_create(&rs[i], NULL, read\_file, new int(i)); // запускаем четный поток

if (res != 0) // проверяем

{

perror("Thread creation failed");

exit(EXIT\_FAILURE);

}

}

int er=-1;

// проверяем завершение потоков, а так же удаляем файлы и очищаем память

for(i=qnt-1; i>=0; i--)

{

tmp = i\*2+1;

res = pthread\_join(ws[i], NULL);

if (res == 0){er = 0;}// printf("Picked up a thread #%i\n", tmp);

else perror("pthread\_join failed");

tmp = i\*2+2;

res = pthread\_join(rs[i], NULL);

if (res == 0){er = 0;} //printf("Picked up a thread #%i\n", tmp);

else perror("pthread\_join failed");

fl = fopen(files[i], "r");

if( fl )

{

fclose(fl);

remove(files[i]);

}

free (files[i]);

}

free (files);

free (ws);

free (rs);

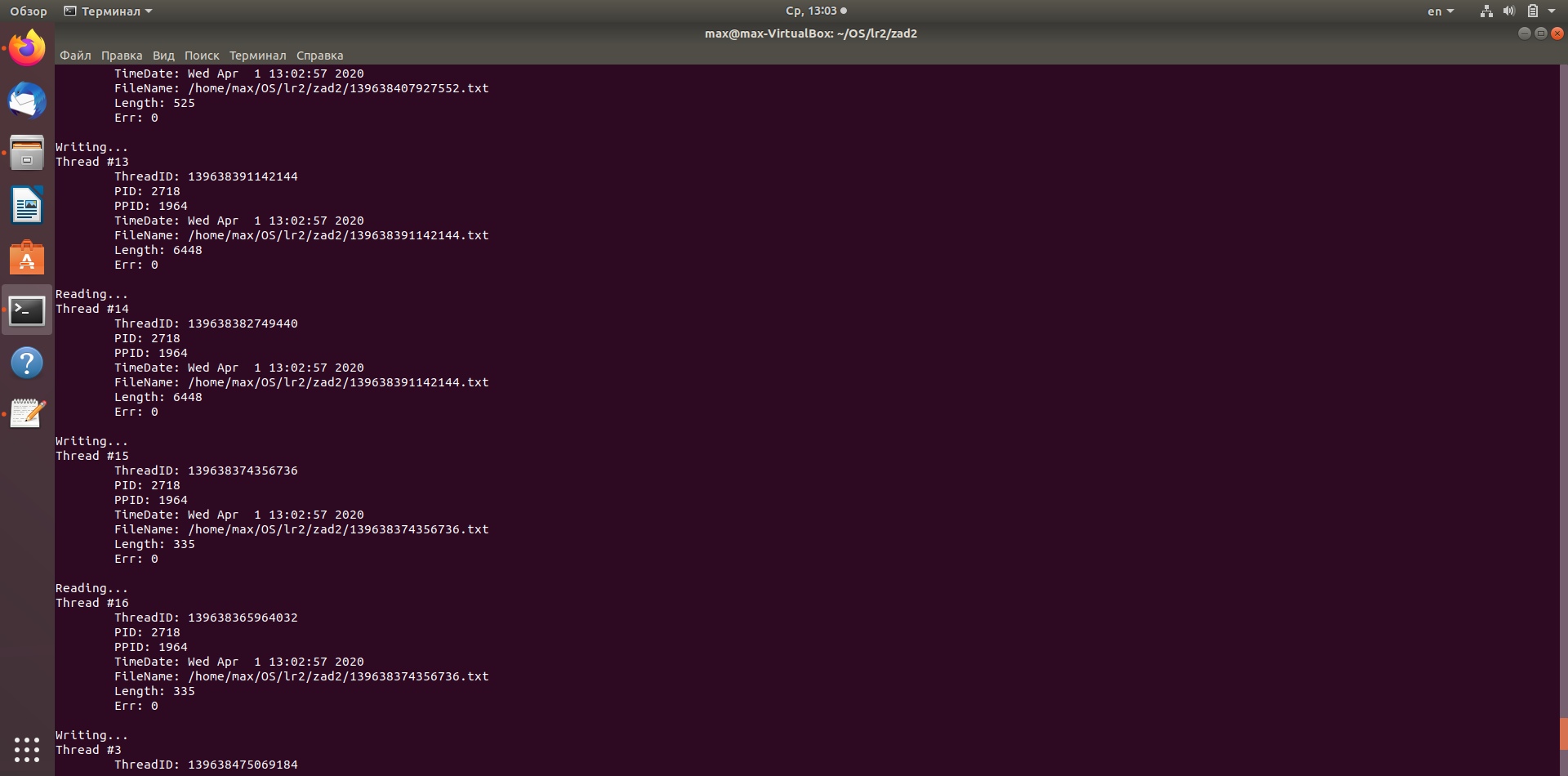
//}

printf("All done\n");

exit(EXIT\_SUCCESS);

}

**Пример выполнения программы:**



**2.3. Ответ на вопрос: Какие атрибуты потоков реализованы в библиотеке Pthreads.**

Функция **pthread\_attr\_init** - инициализирует объект атрибутов потока:

#include <pthread.h>  
int pthread\_attr\_init(pthread\_attr\_t \*attr);

Функция для уничтожения**: pthread\_attr\_destroy**. Ее задача — обеспечить чистое уничтожение объекта атрибутов. После того как объект уничтожен, он не может быть использован снова до тех пор, пока не будет инициализирован повторно.

**detachedstate** — этот атрибут позволяет избежать необходимости присоединения потоков (rejoin). Как и большинство этих функций с префиксом \_set, эта функция принимает указатель на атрибут и флаг для определения требуемого состояния. Два возможных значения флага для функции **attr\_setdetachstate** — **PTHREAD\_CREATE\_JOINABLE** и **PTHREAD\_CREATE\_DETACHED**. По умолчанию у атрибута будет значение **PTHREAD\_CREATE\_JOINABLE**, поэтому можно разрешить двум потокам объединяться (один ждет завершения другого). Если задать состояние **PTHREAD\_CREATE\_DETACHED**, нельзя будет вызвать функцию pthread\_join, чтобы выяснить код завершения другого потока.

**schedpolicy** — этот атрибут управляет планированием потоков. Возможные значения —**SCHED\_OTHER**, **SCHED\_RR** и **SCHED\_FIFO**. По умолчанию атрибут равен SCHED\_OTHER. Два других типа планирования доступны только для процессов, выполняющихся с правами суперпользователя, поскольку они оба задают планирование в режиме реального времени, но с немного разным поведением. SCHED\_RR использует круговую или циклическую схему планирования, a SCHED\_FIFO — алгоритм "первым прибыл, первым обслужен". Оба эти алгоритма не обсуждаются в этой книге.

**schedparam** — это напарник атрибута schedpolicy и позволяет управлять планированием потоков, выполняющихся с типом планирования SCHED\_OTHER.

**inheritsched** — этот атрибут принимает одно из двух значений: **PTHREAD\_EXPLICIT\_SCHED** и **PTHREAD\_INHERIT\_SCHED**. По умолчанию значение атрибута PTHREAD\_EXPLICIT\_SCHED, что означает планирование, явно заданное атрибутами. Если задать PTHREAD\_INHERIT\_SCHED, новый поток будет вместо этого применять параметры, используемые потоком, создавшим его.

**scope** — этот атрибут управляет способом вычисления параметров планирования потока. ОС Linux в настоящее время поддерживает единственное значение **PTHREAD\_SCOPE\_SYSTEM**.

**stacksize** — этот атрибут управляет размером стека при создании потока, задается в байтах. Это часть необязательного раздела стандарта и поддерживается только в тех реализациях, у которых определено значение **\_PTHREAD\_THREAD\_ATTR\_STACKSIZE**. Linux по умолчанию реализует потоки со стеком большого размера, поэтому этот атрибут в ОС Linux избыточен.