МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ   
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ**

**ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»**

**(БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Лабораторная работа №8

Дисциплина: Операционные системы

Тема:

«Многопоточное программирование и средства

синхронизации процессов в ОС Linux»

|  |  |
| --- | --- |
|  | Выполнил:  студента 3 курса  Направления подготовки  09.03.04 –  Программная инженерия  группы ПВ-31  Ковалев П.А. |
|  | Проверил:  Дмитренко П.С.  «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г.  Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Подпись: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

Белгород 2020 г.

**Цель работы:** ознакомиться с принципами создания многопоточных приложений и основными элементами синхронизации процессов и потоков.

**Задания к лабораторной работе**

1. Разработайте программу, реализующую механизм управления процессами.

В программе должны быть реализованы следующие функции:

(а) Создать процесс

(b) Завершить созданный процесс

(с) Дождаться завершения процесса

1. Создайте многопоточное приложение по следующему описанию
2. Запустить 5 параллельных потоков. Каждый поток должен вывести на экран свой номер. Программа должна ожидать завершения всех потоков
3. Реализуйте внутри потоков алгоритм суммирования единиц:

Каждый поток должен прибавлять в цикле единицу к двум переменным

* + Общей переменной для всех потоков
  + Частной переменной, которая у каждого потока своя

После выполнения всего цикла суммирования значения всех суммируемых переменных необходимо вывести их на экран. Охарактеризуйте полученные результаты.

1. Примените mutex для синхронизации потоков
2. Реализуйте в программе механизм запуска только одной копии процесса

(b) Реализуйте идею критических секций для алгоритма суммирования из

предыдущего этапа лабораторной работы таким образом, чтобы программа получала верные результаты.

**Выполнение**

**Код программы**

#define COUNT\_THREAD 5

#include <iostream>

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#include <wait.h>

using namespace std;

typedef struct

{

int\* sum\_all\_t3;

pthread\_mutex\_t\* mutex;

} data\_t3;

int main()

{

task\_one();

//task\_two();

//task\_three();

system("pause");

return 0;

}

void task\_one()

{

bool flag = true;

int run\_command = 1, pid = 1, tmp\_pid;

cout << "List command:" << endl;

cout << "0 - exit" << endl;

cout << "1 - fork" << endl;

cout << "2 - wait" << endl;

cout << "3 - waitpid" << endl;

cout << "4 - kill" << endl;

while (flag)

{

if (run\_command)

{

cout << "Input command:" << endl;

cin >> run\_command;

}

else

{

return;

}

switch (run\_command)

{

case 1:

pid = fork();

if (pid)

cout << "pid = " << pid << endl;

break;

case 2:

cout << "wait" << endl;

wait(0);

break;

case 3:

cout << "Input pid for waitpid: ";

cin >> tmp\_pid;

waitpid(tmp\_pid, 0, 0);

break;

case 4:

cout << "Input pid for kill: ";

cin >> tmp\_pid;

kill(tmp\_pid, 9);

break;

default:

flag = false;

break;

}

}

}

void\* thread\_f2(void\* arg)

{

int sum\_this\_thread = 0;

int\* sum\_all\_thread = (int\*)arg;

pthread\_t pid = pthread\_self(); //возвращаем идентификатор потока, в котором вызывает функция

printf("PID this thread = %d\n", pid);

for (int i = 0; i < 1000; ++i)

{

++sum\_this\_thread; //прибавляем 1 к сумме этого потока

++(\*sum\_all\_thread); //прибавляем 1 к общей сумме потоков

}

printf("Sum this thread = %d\n", sum\_this\_thread);

printf("Sum all thread = %d\n", (\*sum\_all\_thread));

printf("Thread %d is done\n\n", pid);

pthread\_exit(nullptr);

}

void task\_two()

{

pthread\_t threads[COUNT\_THREAD]; //массив идентификаторов потока

int\* arg = new int;

\*arg = 0;

for (int i = 0; i < COUNT\_THREAD; ++i) //создаем 5 потоков

pthread\_create(&(threads[i]), NULL, thread\_f2, arg);

for (pthread\_t pid : threads) //ожидаем завершения всех потоков

pthread\_join(pid, 0);

}

void\* thread\_f3(void\* arg)

{

int sum\_this\_thread = 0;

data\_t3 data = \*(data\_t3\*)arg;

int\* sum\_all\_thread = data.sum\_all\_t3;

pthread\_mutex\_t\* mutex = data.mutex;

pthread\_t pid = pthread\_self(); //возвращаем идентификатор потока, в котором вызывает функция

printf("PID this thread = %d\n", pid);

for (int i = 0; i < 20000; ++i)

{

++sum\_this\_thread; //прибавляем 1 к сумме этого потока

//занять мьютекс, если он не занят, иначе ждать его освобождения

pthread\_mutex\_lock(mutex);

++(\*sum\_all\_thread); //прибавляем 1 к общей сумме потоков

//освободить мьютекс

pthread\_mutex\_unlock(mutex);

}

printf("Sum this thread = %d\n", sum\_this\_thread);

printf("Sum all thread = %d\n", (\*sum\_all\_thread));

printf("Thread %d is done\n\n", pid);

pthread\_exit(0);

}

void task\_three()

{

pthread\_t threads[COUNT\_THREAD];

data\_t3\* arg = new data\_t3;

(\*arg).sum\_all\_t3 = new int;

(\*(\*arg).sum\_all\_t3) = 0;

(\*arg).mutex = new pthread\_mutex\_t;

pthread\_mutex\_init((\*arg).mutex, 0);

for (int i = 0; i < COUNT\_THREAD; ++i) //создаем 5 потоков

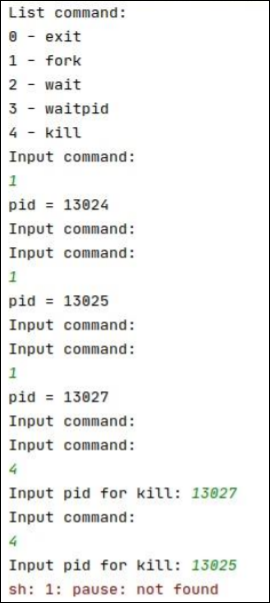
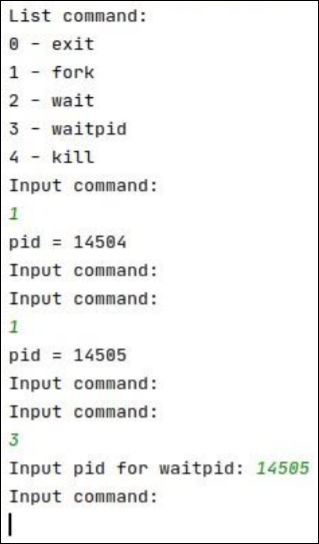
pthread\_create(&(threads[i]), NULL, thread\_f3, arg);

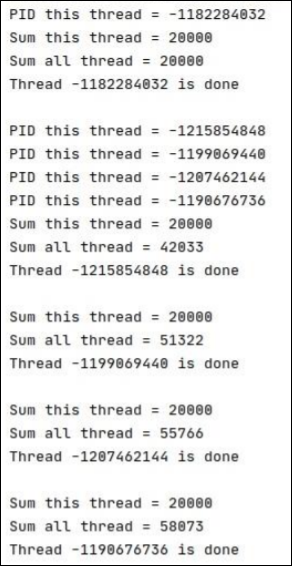
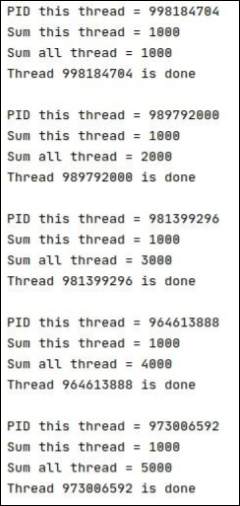
for (pthread\_t pid : threads) //ожидаем завершения всех потоков

pthread\_join(pid, 0);

}

**Task\_one**

**Task\_two**

Исходя из результатов программы при разном диапазоне в цикле, можно сделать вывод, что при небольших значениях числа повторения операций результат суммирования обеих переменных будет верным, но, чем больше число повторений операций, тем больше вероятность неверного результата в переменной суммы операций всех потоков.

**Task\_three**

Исходя из результатов работы программы в разных диапазонах числа операций, можно сделать вывод, что благодаря мьютексу ошибки в расчете общей суммы числа операций потоков не будет вне зависимости от числа повторяемых операций.

(с) Продемонстрируйте пример возникновения взаимной блокировки в вашей программе.

Замечания: внесите по возможности минимальные модификации в программу для возникновения взаимных блокировок.

Взаимная блокировка возникает в случае, когда мы заняли мьютекс (pthread\_mutex\_lock()) одним из потоков, но не освободили его для других.

В результате ни один из потоков не может закончить выполнение операции. Чтобы этого избежать, необходимо освободить мьютекс командой pthread\_mutex\_unlock().

|  |  |
| --- | --- |
| void\* thread\_f3(void\* arg)  {  int sum\_this\_thread = 0;  data\_t3 data = \*(data\_t3\*)arg;  int\* sum\_all\_thread = data.sum\_all\_t3;  pthread\_mutex\_t\* mutex = data.mutex;  pthread\_t pid = pthread\_self();  printf("PID this thread = %d\n", pid);  for (int i = 0; i < 20000; ++i)  {  ++sum\_this\_thread; //прибавляем 1 к сумме этого потока  //занять мьютекс, если он не занят, иначе ждать его освобождения  pthread\_mutex\_lock(mutex);  ++(\*sum\_all\_thread); //прибавляем 1 к общей сумме потоков  //освободить мьютекс  //pthread\_mutex\_unlock(mutex);  }  printf("Sum this thread = %d\n", sum\_this\_thread);  printf("Sum all thread = %d\n", (\*sum\_all\_thread));  printf("Thread %d is done\n\n", pid);  pthread\_exit(0);  } |  |

**Контрольные вопросы**

1. Что такое взаимная блокировка? При каких условиях она возникает?

Взаимная блокировка – блокировка действий для всех потоков.

Она возникает в случае, когда мы заняли мьютекс (pthread\_mutex\_lock()) одним из потоков, но не освободили его для других. В результате ни один из потоков не может закончить выполнение операции. Чтобы этого избежать, необходимо освободить мьютекс командой pthread\_mutex\_unlock().

2. Что такое мьютекс/семафор?

Семафор – это механизм, который позволяет конкурирующим процессам и потокам работать с общими ресурсами и помогает в решении различных проблем синхронизации. Мьютекс – механизм синхронизации, который, по сути, является бинарным семафором (имеет два состояния: «занят», «не занят»).

3. Для чего применяется функция pthread\_join?

Для ожидания завершения потока. Без нее многопоточная программа завершится, когда до конца выполнится один из потоков.