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 №4

По предмету «Операционные системы и системное ПО»

«Синхронизация потоков»

Выполнил:

студент группы ПО(аб) –81

Чекулаев В.Ю.

Проверил :

Тормозов В.С.

Хабаровск – 2021г.

**Цели**: Изучение функций, предназначенных для синхронизации потоков в ОС Linux.

**Задачи**: Получение практических навыков использования мьютексов и условных переменных для решения задач доступа к ресурсам.

**Порядок выполнения работы:**

1. Выполните следующие действия. Для каждого изменения опишите, что происходит, и дайте объяснение – почему.

1) Уберите в первом примере функцию sleep.

Текст первого примера:

#include <pthread.h>

#include <stdio.h>

int x=1;

void\* compute\_thread(void \* argument)

{

printf("X value in thread before sleep = %d\n",x);

printf("X value in thread is incremented by 1 before sleep\n");

x++;

sleep(2);

printf("X value in thread after sleep = %d\n",x);

return;

}

main( )

{

pthread\_t tid;

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);

pthread\_create(&tid, &attr, compute\_thread,(void \*)NULL);

sleep(1);

x++;

printf("Main thread increments X, after that X value is %d\n",x);

pthread\_join(tid,NULL);

exit(0);

}

Единственный способ приостановки потока — сделать это из кода самого потока. Другими словами, можно только приостановить текущий поток. Что бы это сделать, можно вызвать функцию sleep().

После создания потока в функции main() следом идет функция sleep(1). После вызова этой функции программа начинает работать в таком порядке:

printf("X value in thread before sleep = %d\n",x);

printf("X value in thread is incremented by 1 before sleep\n");

x++;

После этого вызывается функция sleep(2), и мы переходим к следующей части кода:

x++;

printf("Main thread increments X, after that X value is %d\n",x);

После приостановка потока завершается и вызывается следующий фрагмент кода:

x++;

printf("Main thread increments X, after that X value is %d\n",x);

Поэтому результат работы программы с вызовами функции sleep будет следующим:

X value in thread before sleep = 1

X value in thread is incremented by 1 before sleep

Main thread increments X, after that X value is 3

X value in thread after sleep = 3

Если функции sleep убрать, то приостановка потока совершаться не будет и результат будет следующим:

Main thread increments X, after that X value is 2

X value in thread before sleep = 2

X value in thread is incremented by 1 before sleep

X value in thread after sleep = 3

2) Уберите во втором примере функцию освобождения мьютекса из порожденного потока.

Текст второго примера:

#include <pthread.h>

#include <stdio.h>

void\* compute\_thread (void\*);

pthread\_mutex\_t my\_sync;

pthread\_cond\_t rx;

int thread\_done = 0;

int x=1;

main( )

{

pthread\_t tid;

pthread\_attr\_t attr;

pthread\_attr\_init (&attr);

pthread\_mutex\_init (&my\_sync, NULL);

pthread\_cond\_init (&rx, NULL);

pthread\_create(&tid, &attr, compute\_thread, (void \*)NULL);

pthread\_mutex\_lock(&my\_sync);

while (!thread\_done) pthread\_cond\_wait(&rx, &my\_sync);

x++;

printf("Main thread increments X, after that X value is %d\n",x);

pthread\_mutex\_unlock(&my\_sync);

exit(0);

}

void\* compute\_thread(void\* dummy)

{

printf("X value in thread before sleep = %d\n",x);

printf("X value in thread is incremented by 1 before sleep\n");

pthread\_mutex\_lock (&my\_sync);

x++;

sleep(2);

5printf("X value in thread after sleep = %d\n",x);

thread\_done = 1;

pthread\_cond\_signal (&rx);

pthread\_mutex\_unlock (&my\_sync);

return;

}

Мьютекс захватывается вызовом функции **pthread\_mutex\_lock**. Он освобождается вызовом функции **pthread\_mutex\_unlock**. **pthread\_mutex\_lock** или захватывает мьютекс, или приостанавливает выполнение вызывающего потока, пока владелец мьютекса (то есть поток, который захватил мьютекс вызовом функции **pthread\_mutex\_lock** ранее) не освободит его вызовом **pthread\_mutex\_unlock**. Так как взаимная блокировка не была освобождена, программа не может завершиться.

3) Уберите в третьем примере функцию pthread\_cond\_signal.

Функция pthread\_cond\_wait() используется, чтобы атомарно освободить мьютекс и заставить вызывающий поток блокироваться по переменной состояния. Блокированный поток пробуждается с помощью вызовов pthread\_cond\_signal(). Процедура pthread\_cond\_wait() всегда возвращает запертый мьютекс, который принадлежит вызывающему потоку, даже если возникла ошибка. Эта функция блокируется, пока не придет сообщение о нужном состоянии. Она атомарно освобождает связанный с ней закрытый мьютекс перед блокированием, и атомарно захватывает его снова, перед возвратом. Чтобы разблокировать определенный поток, используется функция pthread\_cond\_signal(). Если никакие потоки не блокированы по переменной состояния, вызов pthread\_cond\_signal () не будет иметь никакого эффекта. Таким образом, без функции pthread\_cond\_signal поток не может быть разблокирован, и программа не может завершиться.

2. Решите классическую проблему «поставщик – потребитель» с использованием описанных в лабораторной работе средств синхронизации.

Постановка задачи: Один поток производит данные, другой поток их потребляет. В промежуток времени между изготовлением и потреблением данные хранятся в буфере.

Пример использования: Конвейер команд в Unix.

Исходные данные: Данные хранятся в циклическом буфере. Циклический буфер описывается некоторой областью памяти, указателем начала данных и указателем конца данных. Поток-поставщик записывает данные в конец буфера, поток-потребитель считывает их с начала буфера. После записи или чтения соответствующим образом меняются указатели начала и конца.

Операции чтения/записи должны быть выполнены как взаимоисключающие. Если операция чтения выполняется над пустым буфером (указатель начала = указатель конца), поток-потребитель должен быть заблокирован на условной переменной до тех пор, пока поток-поставщик не запишет в буфер какие-нибудь данные. Если операция записи выполняется над полным буфером, поток-поставщик должен также быть заблокирован на условной переменной до тех пор, пока поток-потребитель не считает из буфера какие-нибудь данные.

Размер буфера – не менее 10 символов. Поток-поставщик и поток-потребитель работают в бесконечном цикле. Поток-поставщик производит по одному символу в последовательности 0,1,2...9,0,1,... и записывает его в буфер через случайный интервал времени 0,5 – 2 сек. Поток-потребитель считывает по одному символу через случайный интервал времени 0,5 – 2 сек из буфера и выводит их на экран в виде сообщений (например, Символ 0, Символ 1,...)

Каждый поток совершая операцию с буфером выводит на экран информацию о текущем состоянии буфера до и после операции, тип операции, символ, состояние условной переменной.

Текст программы:

#include <pthread.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <iostream>

#include <string>

using namespace std;

void\* write\_task(void\*);

void\* read\_task(void\*);

bool buffer\_init(int);

void print\_str\_info(int, bool, int);

struct buffer

{

int size; // общий размер

int amount; // количество записанных символов

int\* start; // начало буфера

int\* end; // конец буфера

int\* data; // указатель на начало буфера

} buf;

pthread\_mutex\_t buf\_mutex;

pthread\_cond\_t buf\_cond;

pthread\_cond\_t buf\_size\_sync\_overwrite;

pthread\_cond\_t buf\_size\_sync\_equal;

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

if(argc != 2){

cout << "One parameter needed!\n";

exit(EXIT\_FAILURE);

}

int buffer\_size = atoi(argv[1]);

if(buffer\_size < 3 || buffer\_size > 50){

cout << "Incorrect buffer size!\n";

cout << "Correct: 3-50\n";

exit(EXIT\_FAILURE);

}

pthread\_t read\_thread;

pthread\_t write\_thread;

pthread\_attr\_t attr;

int res = pthread\_attr\_init(&attr);

if(res != 0){

cout << "Thread attribute initialization failed!\n";

exit(EXIT\_FAILURE);

}

res = pthread\_mutex\_init(&buf\_mutex, NULL);

if(res != 0){

cout << "Buf mutex initialization failed!\n";

exit(EXIT\_FAILURE);

}

res = pthread\_cond\_init(&buf\_cond, NULL);

if(res != 0){

cout << "Buf cond var initialization failed!\n";

exit(EXIT\_FAILURE);

}

res = pthread\_cond\_init(&buf\_size\_sync\_overwrite, NULL);

if(res != 0){

cout << "Buf size sync overwrite cond var initialization failed!\n";

exit(EXIT\_FAILURE);

}

res = pthread\_cond\_init(&buf\_size\_sync\_equal, NULL);

if(res != 0){

cout << "Buf size sync equal cond var initialization failed!\n";

exit(EXIT\_FAILURE);

}

if(!buffer\_init(buffer\_size-1)){

return EXIT\_FAILURE;

}

res = pthread\_create(&write\_thread, &attr, write\_task, (void\*)NULL);

if(res != 0){

cout << "Write thread creation failed!\n";

exit(EXIT\_FAILURE);

}

res = pthread\_create(&read\_thread, &attr, read\_task, (void\*)NULL);

if(res != 0){

cout << "Read thread creation failed!\n";

exit(EXIT\_FAILURE);

}

res = pthread\_join(write\_thread, NULL);

if(res != 0){

cout << "Thread join failed!\n";

}

return EXIT\_SUCCESS;

}

bool buffer\_init(int size){

if(size > 0){

buf.size = size;

buf.amount = 0;

buf.start = new int[buf.size];

buf.end = buf.start;

buf.data = buf.start;

}else{

cout << "Can not initialize buffer!\n";

return false;

}

return true;

}

void\* write\_task(void\* arg){

int counter = 0;

int elems\_counter = 0;

int pos = 0;

while(true){

pthread\_mutex\_lock(&buf\_mutex);

\*buf.start = counter;

pos = elems\_counter;

if(buf.amount == buf.size){

pthread\_cond\_wait(&buf\_size\_sync\_overwrite, &buf\_mutex);

}else if(elems\_counter == buf.size){

buf.start -= buf.size;

elems\_counter = 0;

}else{

++elems\_counter;

++buf.start;

++buf.amount;

pthread\_cond\_signal(&buf\_size\_sync\_equal);

}

pthread\_mutex\_unlock(&buf\_mutex);

print\_str\_info(counter, true, pos);

if(counter == 9){

counter = 0;

}else{

++counter;

}

usleep(rand() % 1500000 + 500000);

}

}

void\* read\_task(void\* arg){

int current\_symbol;

int elems\_counter = 0;

int pos = 0;

usleep(10000);

while(true){

pthread\_mutex\_lock(&buf\_mutex);

if(buf.amount == 0){

pthread\_cond\_wait(&buf\_size\_sync\_equal, &buf\_mutex);

}

current\_symbol = \*buf.end;

pos = elems\_counter;

if(elems\_counter == buf.size){

pthread\_cond\_signal(&buf\_size\_sync\_overwrite);

buf.end -= buf.size;

elems\_counter = 0;

}else{

++elems\_counter;

++buf.end;

--buf.amount;

}

pthread\_mutex\_unlock(&buf\_mutex);

print\_str\_info(current\_symbol, false, pos);

usleep(rand() % 1500000 + 500000);

}

}

void print\_str\_info(int symbol, bool mode, int current\_pos){ //write - true, read - false

if(mode){

//out << "Write thread::symbol wrote - " << symbol << " \n";

cout << " " << " ";

for(int i = 0; i <= buf.size; ++i){

if(i == current\_pos){

cout << "\*";

}else{

cout << buf.data[i];

}

}

cout << " W - " << symbol << "\n";

}else{

//cout << " " << "Read thread::symbol readed - " << symbol << "\n";

cout << "R - " << symbol << " ";

for(int i = 0; i <= buf.size; ++i){

if(i == current\_pos){

cout << ".";

}else{

cout << buf.data[i];

}

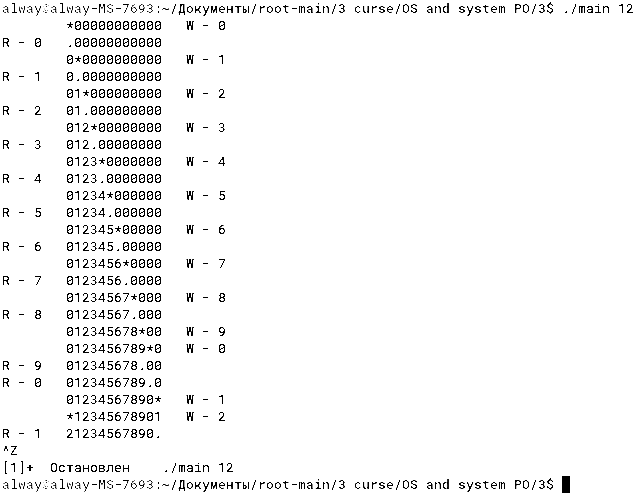
}

cout << "\n";

}

}

Результат работы программы: в цикле записываются и считываются символы от 1 до 9.



Вывод: в ходе лабораторной работы были рассмотрены такие понятия как мьютексы и условные переменные, а также изучены функции для синхронизации потоков.