Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики

Факультет Программной инженерии и компьютерной техники

**Лабораторная работа №2**

по дисциплине

«Многопоточное программирование»

**Выполнил**

Ореховский А.,

группа P3417

**Преподаватель**

Доронин О. В.

Санкт-Петербург

2021

# Листинг кода

#include <pthread.h>

#include <cstdlib>

#include <vector>

#include <string>

#include <iostream>

#include <atomic>

#include <map>

#include <cassert>

#include <sstream>

#include <unistd.h>

#include <chrono>

struct consume\_args\_t

{

    pthread\_mutex\_t\* mutex;

    bool is\_debug;

    bool\* exit;

    bool\* updated;

    long long int\* value;

    int sleep;

    void set(

        pthread\_mutex\_t\* mutex,

        bool is\_debug,

        bool\* exit,

        bool\* updated,

        long long int\* value,

        int sleep)

    {

        this->mutex = mutex;

        this->is\_debug = is\_debug;

        this->exit = exit;

        this->updated = updated;

        this->value = value;

        this->sleep = sleep;

    }

};

struct produce\_args\_t

{

    pthread\_mutex\_t\* mutex;

    bool\* updated;

    bool\* exit;

    long long int\* value;

    int consume\_threads\_count;

    produce\_args\_t(

        pthread\_mutex\_t\* mutex,

        bool\* updated,

        bool\* exit,

        long long\* value,

        const int consume\_threads\_count)

        :

        mutex(mutex),

        updated(updated),

        exit(exit), value(value),

        consume\_threads\_count(consume\_threads\_count)

    {

    }

};

struct interrupt\_args\_t

{

    int consume\_threads\_count;

    bool\* exit;

    std::vector<pthread\_t>\* threads;

    interrupt\_args\_t(

        bool\* exit,

        const int consume\_threads\_count,

        std::vector<pthread\_t> \*threads)

        :

        consume\_threads\_count(consume\_threads\_count),

        exit(exit),

        threads(threads)

    {

    }

};

int run\_treads(int consume\_threads\_count, int delay\_time, bool is\_debug);

int get\_tid();

void\* producer\_routine(void\* arg);

void\* consumer\_routine(void\* arg);

void\* consumer\_interrupter\_routine(void\* arg);

int read\_numbers(const std::string& s, std::vector<long long>& v);

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

{

    auto start\_time = std::chrono::high\_resolution\_clock::now();

    int consume\_threads\_count\_pos, delay\_time\_pos;

    auto is\_debug = false;

    if (argc == 4 && std::string(argv[1]) == "-debug")

    {

        is\_debug = true;

        consume\_threads\_count\_pos = 2;

        delay\_time\_pos = 3;

    }

    else if (argc == 4 && std::string(argv[3]) == "-debug")

    {

        is\_debug = true;

        consume\_threads\_count\_pos = 1;

        delay\_time\_pos = 2;

    }

    else if (argc == 3)

    {

        consume\_threads\_count\_pos = 1;

        delay\_time\_pos = 2;

    }

    else

    {

        return 1;

    }

    const auto consume\_threads\_count = strtol(argv[consume\_threads\_count\_pos], nullptr, 10);

    auto delay\_time = strtol(argv[delay\_time\_pos], nullptr, 10);

    if (consume\_threads\_count < 1 || delay\_time < 0)

    {

        return 1;

    }

    std::cout << run\_treads(consume\_threads\_count, delay\_time, is\_debug) << std::endl;

    auto end\_time = std::chrono::high\_resolution\_clock::now();

    std::cout << "Time difference:" << std::chrono::duration\_cast<std::chrono::milliseconds>(end\_time - start\_time).count() << " milliseconds" << std::endl;

}

int run\_treads(const int consume\_threads\_count, int delay\_time, const bool is\_debug)

{

    pthread\_mutex\_t mutex;

    auto updated = false, exit = false;

    long long int value = 0;

    std::vector<pthread\_t> threads(consume\_threads\_count + 2);

    delay\_time \*= 1000;

    std::vector<consume\_args\_t> consume\_arg(consume\_threads\_count);

    assert(pthread\_mutex\_init(&mutex, nullptr) == 0);

    for (auto i = 0; i < consume\_threads\_count; i++)

    {

        consume\_arg[i].set(&mutex, is\_debug, &exit, &updated, &value, delay\_time);

        pthread\_create(&threads[i + 2], nullptr, consumer\_routine, static\_cast<void\*>(&consume\_arg.at(i)));

    }

    produce\_args\_t produce\_arg(&mutex, &updated, &exit, &value, consume\_threads\_count);

    pthread\_create(&threads[0], nullptr, producer\_routine, static\_cast<void\*>(&produce\_arg));

    interrupt\_args\_t interrupt\_arg(&exit, consume\_threads\_count, &threads);

    pthread\_create(&threads[1], nullptr, consumer\_interrupter\_routine, static\_cast<void\*>(&interrupt\_arg));

    auto result = 0, sum = 0;

    pthread\_join(threads[0], nullptr);

    pthread\_join(threads[1], nullptr);

    for (auto i = 0; i < consume\_threads\_count; i++)

    {

        pthread\_join(threads[i + 2], reinterpret\_cast<void\*\*>(&result));

        sum += result;

    }

    return sum;

}

int get\_tid()

{

    static std::atomic<int> threads\_count;

    thread\_local int id = threads\_count++;

    return id;

}

void\* producer\_routine(void\* arg)

{

    auto\* produce\_arg = static\_cast<produce\_args\_t\*>(arg);

    std::string str;

    std::vector<long long> numbers;

    std::getline(std::cin, str);

    read\_numbers(str, numbers);

    std::string line\_part;

    for (auto number: numbers)

    {

        while (\*produce\_arg->updated)

        {

            usleep(0);

        }

        \*produce\_arg->value = number;

        \*produce\_arg->updated = true;

    }

    \*produce\_arg->exit = true;

    pthread\_exit(nullptr);

    return nullptr;

}

void\* consumer\_routine(void\* arg)

{

    pthread\_setcancelstate(PTHREAD\_CANCEL\_DISABLE, NULL);

    auto\* consume\_arg = static\_cast<consume\_args\_t\*>(arg);

    long long int sum = 0;

    while (!(\*consume\_arg->exit) || \*consume\_arg->updated)

    {

        if (\*consume\_arg->updated)

        {

            pthread\_mutex\_lock(consume\_arg->mutex);

            if (\*consume\_arg->updated)

            {

                sum += \*consume\_arg->value;

                \*consume\_arg->updated = false;

                pthread\_mutex\_unlock(consume\_arg->mutex);

                if (consume\_arg->is\_debug)

                {

                    std::cout << "(" + std::to\_string(get\_tid()) +  ", " + std::to\_string(sum) + ")\n";

                }

            }

            else

            {

                pthread\_mutex\_unlock(consume\_arg->mutex);

            }

        }

        usleep(100);

        usleep(consume\_arg->sleep == 0 ? 0 : rand() % consume\_arg->sleep);

    }

    pthread\_exit(reinterpret\_cast<void\*>(sum));

    return nullptr;

}

void\* consumer\_interrupter\_routine(void\* arg)

{

    auto\* interrupt\_arg = static\_cast<interrupt\_args\_t\*>(arg);

    while (\*interrupt\_arg->exit == false)

    {

        pthread\_cancel((\*interrupt\_arg->threads)[rand() % interrupt\_arg->consume\_threads\_count + 2]);

    }

    pthread\_exit(nullptr);

    return nullptr;

}

int read\_numbers(const std::string& s, std::vector<long long>& v)

{

    std::istringstream is(s);

    int n;

    while (is >> n)

    {

        v.push\_back(n);

    }

    return v.size();

}

# Итоги

После выполнения ЛР, я получил понимание о том, как работают примитивы синхронизации в C++, а так же научился запускать IDE МЫ Code из под WSL.