Факультет программной инженерии и компьютерной техники

Теоретические основы компьютерной графики и вычислительной оптики

Лабораторная работа №2

Преподаватель: Доронин Олег Владимирович

Выполнил: студент: Кульбако Артемий Юрьевич, P4115

Санкт-Петербург
2022

ЗАДАНИЕ

Ваша задача - реализовать классический паттерн producer-consumer с небольшими дополнительными условиями. Программа должна состоять из 3+N потоков:

1. главный
2. producer
3. interruptor
4. N потоков consumer

Так-же необходимо реализовать поддержку ключа –debug, при использовании которого каждый consumer-поток будет выводить пару (tid, psum), где tid реализуется с помощью функции get_tid(), а psum это сумма которую посчитал поток. Вывод значений psum происходит при каждом изменении.

Функция get_tid() возвращает идентификатор потока. Идентификатор потока это не проста pthread_self(), а уникальное для каждого потока число в диапозоне от 1 .. 3+N. Значение этого числа предполагается хранить в TLS. Память под сохраняемое значение должно выделяться в heap, а указатель на него в TLS. Так-же функция get_tid должна быть самодостаточной (для использования ее в другом проекте должно быть достаточно только скопировать get_tid и использовать)

В поток вывода должно попадать только результирующее значение, по умолчанию никакой отладочной или запросной информации выводиться не должно.

ОСНОВОНОЙ КОД ПРОГРАММЫ (producer_consumer.cpp)

```
#include "producer_consumer.h"

static struct {
  bool debug = false;
  int num_of_consumers;
  int sleep_limit;
} PARAMS;

static struct {
  vector<pair<pthread_t, int*>> consumers;
  bool is_tasks_supplied = false;
  pthread_cond_t consumer_cond, producer_cond;
  pthread_mutex_t mutex;
  queue<int> tasks;
} STATE;

int rand_include(int max, int min = 0) {
  srand(time(NULL));
  return (max <= min || abs(max) == abs(min))
             ? 0
             : min + rand() % ((max + 1) - min);
}

int get_tid() {
  static atomic_int last_thread_id(0);
  thread_local int id = -1;
  if (id == -1) id = ++last_thread_id;
  return id;
}

void* producer_routine(void*) {
  while (!STATE.is_tasks_supplied) {
    pthread_mutex_lock(&STATE.mutex);
    if (!STATE.tasks.empty())
      pthread_cond_wait(&STATE.producer_cond, &STATE.mutex);
    string nums_raw;
    getline(cin, nums_raw);
    stringstream is(nums_raw);
    vector<int> nums_parsed((istream_iterator<int>(is)),
                            (istream_iterator<int>()));
    for (auto it : nums_parsed) STATE.tasks.push(it);
    STATE.is_tasks_supplied = true;
    pthread_mutex_unlock(&STATE.mutex);
  }
  bool polling = true;
  while (polling) {
    pthread_mutex_lock(&STATE.mutex);
    if (!STATE.tasks.empty())
      pthread_cond_signal(&STATE.consumer_cond);
    else
      polling = false;
    pthread_mutex_unlock(&STATE.mutex);
  }
  return NULL;
}
```

```
void* consumer_routine(void* arg) {
  int* partitial_sum = (int*)arg;
  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
  while (true) {
    bool task_completed = false;
    pthread_mutex_lock(&STATE.mutex);
    if (STATE.tasks.empty() && !STATE.is_tasks_supplied) {
      pthread_cond_signal(&STATE.producer_cond);
      pthread_cond_wait(&STATE.consumer_cond, &STATE.mutex);
    }
    if (!STATE.tasks.empty()) {
      int candidate = STATE.tasks.front();
      *partitial_sum += candidate;
      STATE.tasks.pop();
      task_completed = true;
      if (PARAMS.debug)
        cout << "(" << get_tid() << ", " << *partitial_sum << ")" << endl;
    }
    if (STATE.is_tasks_supplied && STATE.tasks.empty()) {
      pthread_mutex_unlock(&STATE.mutex);
      break;
    }
    pthread_mutex_unlock(&STATE.mutex);
    if (task_completed)
      usleep(rand_include(PARAMS.sleep_limit) *
             1000);  // convert microsecond to millis
  }
  return NULL;
}

void* consumer_interruptor_routine(void*) {
  while (!STATE.is_tasks_supplied)
    pthread_cancel(
        STATE.consumers[rand_include(0, STATE.consumers.size() -
1)].first);
  return NULL;
}

int run_threads(int num_of_consumers, int sleep_limit, bool debug) {
  PARAMS.num_of_consumers = num_of_consumers;
  PARAMS.sleep_limit = sleep_limit;
  PARAMS.debug = debug;
  pthread_t producer;
  pthread_t interruptor;
  pthread_mutex_init(&STATE.mutex, NULL);
  pthread_cond_init(&STATE.producer_cond, NULL);
  pthread_cond_init(&STATE.consumer_cond, NULL);
  pthread_create(&producer, NULL, producer_routine, NULL);
  for (auto i = 0; i < PARAMS.num_of_consumers; i++) {
    pthread_t consumer;
    int* partitial_sum = new int(0);
    pthread_create(&consumer, NULL, consumer_routine,
(void*)partitial_sum);
    STATE.consumers.push_back(make_pair(consumer, partitial_sum));
  }
  pthread_create(&interruptor, NULL, consumer_interruptor_routine, NULL);
  pthread_join(interruptor, NULL);
  pthread_join(producer, NULL);
  return accumulate(STATE.consumers.begin(), STATE.consumers.end(), 0,
                    [](int acc, pair<pthread_t, int*> it) {
                      pthread_join(it.first, NULL);
                      return acc + *it.second;
                    });
```

```
}
```

ОСНОВНОЙ КОД ТЕСТОВ (tests.cpp)

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include <doctest.h>
#include <producer_consumer.h>
#include <chrono>
#include <fstream>
#include <thread>

using namespace std;

// i knew, it's return cores, not threads, but I think it's ok for testing
const int MAX_THREADS = std::thread::hardware_concurrency();

TEST_CASE("test get_tid()") {
  for (auto i = 1; i <= MAX_THREADS; i++) {
    pthread_t tester;
    pthread_create(&tester, NULL,
                   [](void* arg) -> void* {
                     int* expected_tid = (int*)arg;
                     auto test_tid = get_tid();
                     CHECK(test_tid == *expected_tid);
                     return NULL;
                   },
                   &i);
     pthread_join(tester, NULL);
    pthread_join(tester, NULL);
  }
}

TEST_CASE("test run_threads()") {
  srand(time(NULL));
  auto r = []() { return 1 + (rand() % 256); };
  vector<int> nums(r());
  generate(nums.begin(), nums.end(), [r]() { return r(); });
  auto expected_sum = accumulate(nums.begin(), nums.end(), 0);
  stringstream ss;
  copy(nums.begin(), nums.end(), ostream_iterator<int>(ss, " "));
  ifstream fin("input.in");
  istringstream data(ss.str());
  cin.rdbuf(data.rdbuf());
  auto MAX_SLEEP_MS = 512;
  auto start = chrono::high_resolution_clock::now();
  auto test_sum = run_threads(8, MAX_SLEEP_MS, false);
  auto stop = chrono::high_resolution_clock::now();
  auto duration =
      chrono::duration_cast<chrono::microseconds>(stop - start).count() *
1000;
  CHECK(test_sum == expected_sum);
  CHECK(duration >= MAX_SLEEP_MS * nums.size());
}
```

ВЫВОД

Лабораторная работа оказалось непростой: пришлось вспомнить, как мыслить потоками, разобраться с примитивами синхронизации C++ и при этом не допустить разных ошибок памяти. Тем не менее, я многому научился: решил классическую задачу читателей-потребителей, написал тесты для C++ кода. Попутно, конечно, пришлось решать некоторые странные проблемы, которыми славится язык (к примеру, один и тот же код мог нормально собираться в Windows, но не собираться на macOS, и наоборот, а в случае с Docker-ом генерировать ещё ошибки, разные для разных ОС).

Полный код (включая конфигурационные файлы и файлы для сборки): https://gitlab.se.ifmo.ru/system-software/itmo_winter_2023/02-posix/testpassword