

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**«Київський політехнічний інститут ім. І. Сікорського»**

**Інститут Прикладного Системного Аналізу**

**Кафедра Системного Проектування**

Лабораторна робота №2

З дисципліни «Паралельні обчислення»

Виконав:

Ст. гр. ДА-12

Кракович Павло

**Лабораторна робота №2**

**Дослідження базових операцій з потоками виконання**

**Мета роботи:** Розглянути базові примітиви синхронізації та їх особливості, в залежності від обраної мови програмування. Розглянути підходи до побудови ПЗ з використанням паралелізму та ознайомитися з класичною задачею паралелізму у вигляді пулу потоків.

**Завдання:**

1. Ознайомитися з визначенням поняття пул потоків (thread pool), використовуючи даний методичний посібник, або ж сторонні джерела. Ознайомитися з примітивами синхронізації та проблемами, пов’язаними з ними.

2. Реалізувати власний пул потоків з характеристиками, зазначеними в обраному варіанті. Обов’язкові для кожного варіанту характеристики: пул потоків повинен бути написаним коректно відносно обраної мови програмування, повинен мати можливість коректного завершення своєї роботи (моментально, з покиданням всіх активних задач, так і з завершенням активних задач), можливість тимчасової зупинки своєї роботи, працювати з використанням умовних змінних)

3. Операції ініціалізації та знищення пулу, додавання та вилучення задач в чергу повинні бути безпечними з точки зору паралелізму.

4. Створити програму, котра буде виконувати задачі за обраним варіантом, використовуючи написаний студентом пул потоків. Код відповідальний за додавання задач в пул потоків, та сам пул потоків повинні знаходитися в різних потоках виконання. Задачі в пул потоків повинні додаватися з декількох потоків.

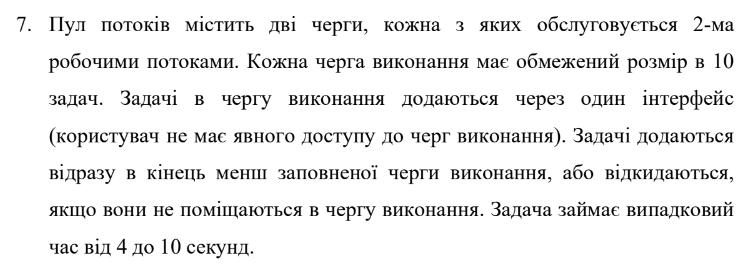
5. Перевірити та довести коректність роботи програми з використанням системи вводу/виводу інформації в консоль (або іншого засобу – профайлера).

6. Зробити обмежене за часом тестування та розрахувати кількість створених потоків та середній час знаходження потоку в стані очікування. Для задач з необмеженою чергою – визначити середню довжину кожної черги та середній час виконання задач. Для обмежених за розміром черг – визначити максимальний та мінімальний час, поки черга була заповнена, кількість відкинутих задач.

7. В протоколі роботи описати ключові моменти реалізації пулу потоків в цілому, а також ключові моменти при імплементації конкретного варіанту завдання. Описати публічний інтерфейс пулу потоків та механізм його роботи. Описати механізм тестування розробленого рішення. Значення метрик, перечисллений в пункті 6 для різних проміжків часу, обраних в довільному порядку самим студентом.

8. Надати висновок, що повинен містити аналіз отриманих результатів.

Варіант роботи №7:

****

**Опис ключових моментів реалізації завдання**

1. Опис ключових моментів загальної реалізації пулу потоків:

1. Створення пулу:

Необхідно розробити код для корректного виконання завдань, які необхідно синхронізовувати, щоб уникнути гонки за доступом, або зайвих очікувань, імплементувати функції по контролю пулу (запуск, пауза, продовження та запуника) та правильно розподілити навантаженнями між ними, якщо це необхідно.

2. Додавання завдань:

Необіхдно розробити контейнер для зберігання задач для забезпечення простого доступу до них. Це може бути черга, або масив. Вони можуть бути відсортовані за пріоритетом, або працювати за принципами FIFO / LIFO.

1. Опис ключових моментів конкретної реалізації пулу потоків:

Для конкретного завдання було написано класс, який містить всі методи роботи тред пулів та всі необхідні змінні, які забезпечують синхронізацію та контроль доступу до даних.

Відповідно до завдання, цей клас містить конструктор, в який передаються черги з головного потоку. Ці самі черги використовуються класом-менеджером завдань для додавання завдань у чергу.

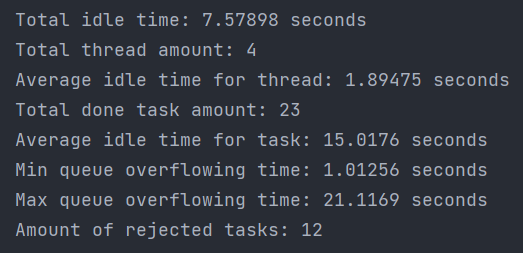
Використовуючи команду запуску та дані, за якими ініціалізувався клас користувач має можливість створити (запустити) тред-пул. В ньому явно прописано створення двох потоків для кожної з переданих черг. При початку роботи з потоками створюється два окремих потоки, які виконують функції для симуляції виконання завдань з черги. Ці потоки мають спільні умовні змінні та мьютекси, що дозволяє їм безпечно, з точки зори паралелізму, виконувати завдання.

Допоміжні функції (pause, resume, stop) забезпечують можливість керувати пулом та працюють за рахунок спільних умовних змінних та спеціальних мьютексів, попарно (start – stop, resume – pause), що забезпечує безпеку виконання завдань.

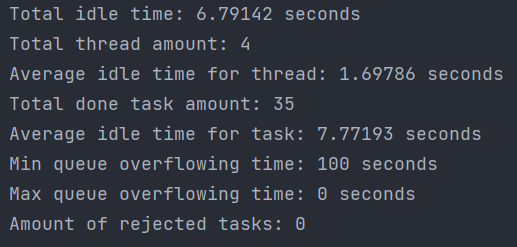
1. Методи тестування розробленого рішення

Для тестування були використані ручні методи у процесі розробки, а саме вивід timestamp для перевірки корретності розрахунку часу та відладка коду у випадках, коли виникали проблеми з синхронізацією, чи доступом. Щоб перевірити надійність коду, кількість задач, які передаються класом-менеджером завдань та їх швидкість додавання є змінною, що дозволяє перевірити роботу розробленного рішення за різних умов.

1. Значення метрик для тестування завдання за різної кількості задач та швидкості їх додавання:
2. 35 задач, задача додається кожну секунду

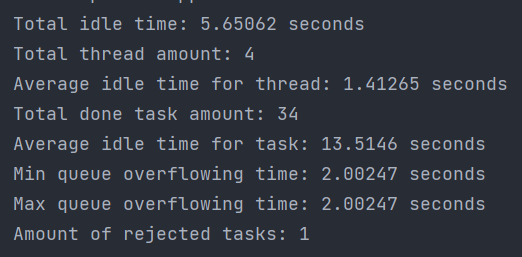


1. 35 задач, задача додається кожні дві секунди



Час переповнення – тестовий, який при виконанні має бути перезаписаний, але цього не відбувається через те, що черга не була переповнена.

1. 35-45 задач, задача додається кожні 1-2 секунди (випадкова кількість)



**Висновок**

Виконавши цю лабораторну роботу, розглянувши базові примітиви синхронізації та їх особливості, розглянувши підходи до побудови ПЗ з використанням паралелізму та ознайомившись з класичною задачею паралелізму у вигляді пулу потоків можна зробити висновок, що пул потоків – це дуже потужний інструмент, який може значно прискорити роботу та виконання задач.

Однак, слід зауважити, що цей метод вимагає розуміння принципів роботи методів синхронизації, принципів роботи умовних змінних для написання моніторів.

Виконавши тестування та переглянувши результати можна зробити висновок, що програма працює корректно, хоча і написано для конкретного завдання та реалізація методів не є універсальною, хоча демонструє можливості концепції моніторів, конструкцій, що дозволяють безпечно працювати паралельним задачам для отримання ексклюзивного доступу до певних даних у програмуванні.

**Додатки**

Код програми:

* main.cpp:
* #include "task\_manager.h"  
  #include "thread\_pool.h"  
  #include <random>  
    
  class TaskManagerThread : public std::thread {  
  public:  
   explicit TaskManagerThread(TaskManager& manager) : std::thread(&TaskManagerThread::run, this, std::ref(manager)) {}  
    
   void run(TaskManager& manager) {  
   for (int i = 1; i <= getRandomValue(35, 45); ++i) {  
   manager.addTask(i);  
   std::this\_thread::sleep\_for(std::chrono::seconds(getRandomValue(1, 2)));  
   }  
   }  
    
  private:  
   static int getRandomValue(int lower\_bound, int upper\_bound) {  
   std::random\_device rd;  
    
   std::uniform\_int\_distribution<int> dist(lower\_bound, upper\_bound);  
    
   return dist(rd);  
   }  
  };  
    
  class ThreadPoolManager {  
  public:  
   explicit ThreadPoolManager(ThreadPool& pool) : thread\_pool(pool) {}  
    
   void run() {  
   thread\_pool.start();  
   }  
    
   void pause() {  
   thread\_pool.pause();  
   }  
    
   void resume() {  
   thread\_pool.resume();  
   }  
    
   void stop() {  
   thread\_pool.stop();  
   }  
    
  private:  
   ThreadPool& thread\_pool;  
  };  
    
  int main() {  
   TaskManager taskManager;  
   ThreadPool threadPool(taskManager.getTaskQueue1(), taskManager.getTaskQueue2());  
   ThreadPoolManager threadPoolManager(threadPool);  
   TaskManagerThread taskManagerThread(taskManager);  
    
   std::this\_thread::sleep\_for(std::chrono::seconds(5));  
   std::thread poolThread(&ThreadPoolManager::run, &threadPoolManager);  
   std::this\_thread::sleep\_for(std::chrono::seconds(5));  
    
   threadPoolManager.pause();  
   std::this\_thread::sleep\_for(std::chrono::seconds(10));  
    
   threadPoolManager.resume();  
   std::this\_thread::sleep\_for(std::chrono::seconds(35));  
    
   if (taskManagerThread.joinable()) {  
   taskManagerThread.join();  
   }  
    
   std::this\_thread::sleep\_for(std::chrono::seconds(5));  
   threadPoolManager.stop();  
    
   if (poolThread.joinable()) {  
   poolThread.join();  
   }  
    
   return 0;  
  }
* task\_queue.h:
* #ifndef TASK\_QUEUE\_H  
  #define TASK\_QUEUE\_H  
    
  #include <queue>  
  #include <shared\_mutex>  
  #include <mutex>  
  #include "timestamp.h"  
    
  using read\_write\_lock = std::shared\_mutex;  
  using read\_lock = std::shared\_lock<read\_write\_lock>;  
  using write\_lock = std::unique\_lock<read\_write\_lock>;  
    
  struct Task {  
   int id;  
   std::chrono::high\_resolution\_clock::time\_point task\_creation\_time;  
  };  
    
  class TaskQueue {  
   using TaskQueueImplementation = std::queue<Task>;  
  public:  
   ~TaskQueue() { clear(); }  
   bool empty() const;  
   size\_t size() const;  
   void clear();  
   bool pop(Task& task);  
   void emplace(int id);  
    
  private:  
   mutable read\_write\_lock m\_rw\_lock;  
   TaskQueueImplementation m\_tasks;  
  };  
    
  bool TaskQueue::empty() const {  
   read\_lock \_(m\_rw\_lock);  
   return m\_tasks.empty();  
  }  
    
  size\_t TaskQueue::size() const {  
   read\_lock \_(m\_rw\_lock);  
   return m\_tasks.size();  
  }  
    
  void TaskQueue::clear() {  
   write\_lock \_(m\_rw\_lock);  
   while (!m\_tasks.empty()) {  
   m\_tasks.pop();  
   }  
  }  
    
  bool TaskQueue::pop(Task& task) {  
   write\_lock \_(m\_rw\_lock);  
   if (m\_tasks.empty()) {  
   return false;  
   } else {  
   task = m\_tasks.front();  
   m\_tasks.pop();  
   return true;  
   }  
  }  
    
  void TaskQueue::emplace(int id) {  
   write\_lock \_(m\_rw\_lock);  
   std::chrono::high\_resolution\_clock::time\_point now = std::chrono::high\_resolution\_clock::now();  
   m\_tasks.emplace(id, now);  
  }  
    
  #endif
* task\_manager.h:
* #ifndef TASK\_MANAGER\_H  
  #define TASK\_MANAGER\_H  
    
  #include "task\_queue.h"  
  #include <iostream>  
    
  class TaskManager {  
  public:  
   TaskManager() {  
   max\_queue\_full\_duration = 0;  
   min\_queue\_full\_duration = 100;  
   abandoned\_tasks = 0.0;  
   current\_duration = 0.0;  
   lastTime = std::chrono::high\_resolution\_clock::now();  
   }  
    
   ~TaskManager() {  
   std::ostringstream task\_manager\_data;  
   task\_manager\_data << "Min queue overflowing time: " << min\_queue\_full\_duration << " seconds" << std::endl  
   << "Max queue overflowing time: " << max\_queue\_full\_duration << " seconds" << std::endl  
   << "Amount of rejected tasks: " << abandoned\_tasks << std::endl;  
   std::cout << task\_manager\_data.str();  
   }  
    
   void addTask(int taskId) {  
   auto currentTime = std::chrono::high\_resolution\_clock::now();  
    
   if (taskQueue1.size() < max\_queue\_size || taskQueue2.size() < max\_queue\_size) {  
   if (current\_duration != 0.0) {  
    
   std::ostringstream queue\_add;  
   queue\_add << Timestamp::getCurrentTimeAsString() << "Tasks can be added again after: "  
   << current\_duration << " seconds" << std::endl;  
   std::cout << queue\_add.str();  
   if (current\_duration > max\_queue\_full\_duration) {  
   max\_queue\_full\_duration = current\_duration;  
   }  
   if (current\_duration < min\_queue\_full\_duration) {  
   min\_queue\_full\_duration = current\_duration;  
   }  
   current\_duration = 0.0;  
   }  
   lastTime = currentTime;  
   if (taskQueue1.size() <= taskQueue2.size()) {  
   taskQueue1.emplace(taskId);  
   std::ostringstream queue\_add1;  
   queue\_add1 << Timestamp::getCurrentTimeAsString() << "Task " << taskId << " added to queue 1" << std::endl;  
   std::cout << queue\_add1.str();  
   } else {  
   taskQueue2.emplace(taskId);  
   std::ostringstream queue\_add2;  
   queue\_add2 << Timestamp::getCurrentTimeAsString() << "Task " << taskId << " added to queue 2" << std::endl;  
   std::cout << queue\_add2.str();  
   }  
   } else {  
   std::ostringstream queue\_full;  
   queue\_full << Timestamp::getCurrentTimeAsString() << "Task " << taskId << " rejected. Queues are full" << std::endl;  
   std::cout << queue\_full.str();  
   abandoned\_tasks++;  
   auto elapsedTime = std::chrono::duration\_cast<std::chrono::duration<double>>(currentTime - lastTime);  
   double timeSeconds = elapsedTime.count();  
   current\_duration += timeSeconds;  
   }  
   }  
    
   TaskQueue& getTaskQueue1() {  
   return taskQueue1;  
   }  
    
   TaskQueue& getTaskQueue2() {  
   return taskQueue2;  
   }  
    
  private:  
   static constexpr int max\_queue\_size = 5;  
   TaskQueue taskQueue1;  
   TaskQueue taskQueue2;  
   double current\_duration;  
   std::chrono::high\_resolution\_clock::time\_point lastTime;  
   double max\_queue\_full\_duration;  
   double min\_queue\_full\_duration;  
   int abandoned\_tasks;  
  };  
    
  #endif
* thread\_pool.h:
* #ifndef THREAD\_POOL\_H  
  #define THREAD\_POOL\_H  
    
  #include "task\_queue.h"  
  #include <iostream>  
  #include <condition\_variable>  
  #include <random>  
    
  class ThreadPool {  
  public:  
   ThreadPool(TaskQueue& queue1, TaskQueue& queue2) : taskQueue1(queue1), taskQueue2(queue2) {}  
    
   ~ThreadPool() {  
   stop();  
   std::ostringstream stop;  
   stop << "Thread pool stopped" << std::endl  
   << "Total idle time: " << totalThreadsIdleTime << " seconds" << std::endl  
   << "Total thread amount: " << poolThreadAmount << std::endl  
   << "Average idle time for thread: " << totalThreadsIdleTime / poolThreadAmount << " seconds" << std::endl  
   << "Total done task amount: " << tasksAmount << std::endl  
   << "Average idle time for task: " << totalTasksIdleTime / tasksAmount << " seconds" << std::endl;  
   std::cout << stop.str();  
   }  
    
   void start() {  
   for (int i = 0; i < 2; ++i) {  
   workers.emplace\_back(std::thread(&ThreadPool::workerFunc, this, std::ref(taskQueue1), std::ref(mutex1), std::ref(cv1)));  
   workers.emplace\_back(std::thread(&ThreadPool::workerFunc, this, std::ref(taskQueue2), std::ref(mutex2), std::ref(cv2)));  
   poolThreadAmount += 2;  
   }  
   }  
    
   void pause() {  
   std::unique\_lock<std::mutex> lock(mutex\_pause);  
   paused = true;  
   std::ostringstream pause;  
   pause << Timestamp::getCurrentTimeAsString() << "Thread pool paused. New tasks will not be taken" << std::endl;  
   std::cout << pause.str();  
   cv1.notify\_all();  
   cv2.notify\_all();  
   }  
    
   void resume() {  
   std::unique\_lock<std::mutex> lock(mutex\_pause);  
   paused = false;  
   std::ostringstream resume;  
   resume << Timestamp::getCurrentTimeAsString() << "Thread pool resumed" << std::endl;  
   std::cout << resume.str();  
   cv1.notify\_all();  
   cv2.notify\_all();  
   }  
    
   void stop() {  
   running = false;  
   cv1.notify\_all();  
   cv2.notify\_all();  
    
   for (auto& worker : workers) {  
   if (worker.joinable()) {  
   worker.join();  
   }  
   }  
   }  
    
  private:  
   void workerFunc(TaskQueue& queue, std::mutex& mutex, std::condition\_variable& cv) {  
   while (running) {  
   std::unique\_lock<std::mutex> thread\_lock(mutex);  
   auto startWaitTime = std::chrono::high\_resolution\_clock::now();  
    
   cv.wait(thread\_lock, [&]() {  
   return !running || !paused || !queue.empty();  
   });  
    
   auto endWaitTime = std::chrono::high\_resolution\_clock::now();  
   auto elapsedWaitTime = std::chrono::duration\_cast<std::chrono::duration<double>>(endWaitTime - startWaitTime);  
    
   if (!paused && !queue.empty()) {  
   Task task;  
   queue.pop(task);  
   thread\_lock.unlock();  
    
   std::chrono::high\_resolution\_clock::time\_point task\_pop\_time = std::chrono::high\_resolution\_clock::now();  
   auto elapsed\_creation = duration\_cast<std::chrono::duration<double>>(task\_pop\_time - task.task\_creation\_time);  
   double task\_queue\_time = elapsed\_creation.count();  
   totalTasksIdleTime += task\_queue\_time;  
   tasksAmount ++;  
    
   std::ostringstream claim;  
   claim << Timestamp::getCurrentTimeAsString() << "Task " << task.id << " claimed. Wait time is: " << task\_queue\_time << " seconds" << std::endl;  
   std::cout << claim.str();  
    
   std::this\_thread::sleep\_for(std::chrono::seconds(getRandomValue(6, 10)));  
    
   std::ostringstream complete;  
   complete << Timestamp::getCurrentTimeAsString() << "Task " << task.id << " completed" << std::endl;  
   std::cout << complete.str();  
   } else {  
   std::lock\_guard<std::mutex> wait\_lock(mutex\_totalIdleTime);  
   totalThreadsIdleTime += elapsedWaitTime.count();  
   }  
   }  
   }  
    
   static int getRandomValue(int lower\_bound, int upper\_bound) {  
   std::random\_device rd;  
   std::uniform\_int\_distribution<int> dist(lower\_bound, upper\_bound);  
   return dist(rd);  
   }  
    
  private:  
   std::vector<std::thread> workers;  
   TaskQueue& taskQueue1;  
   TaskQueue& taskQueue2;  
   std::mutex mutex1;  
   std::mutex mutex2;  
   std::condition\_variable cv1;  
   std::condition\_variable cv2;  
   std::mutex mutex\_pause;  
   double totalThreadsIdleTime = 0.0;  
   double totalTasksIdleTime = 0.0;  
   int poolThreadAmount = 0;  
   int tasksAmount = 0;  
   std::mutex mutex\_totalIdleTime;  
   bool running = true;  
   bool paused = false;  
  };  
    
  #endif
* timestamp.h:
* #ifndef TIMESTAMP\_H  
  #define TIMESTAMP\_H  
    
  #include <iomanip>  
    
  class Timestamp {  
  public:  
   static std::string getCurrentTimeAsString() {  
   auto now = std::chrono::system\_clock::now();  
   std::time\_t time = std::chrono::system\_clock::to\_time\_t(now);  
   struct tm\* ptm = std::localtime(&time);  
   std::ostringstream oss;  
   oss << "[" << std::put\_time(ptm, "%T") << "] ";  
   return oss.str();  
   }  
  };  
    
  #endif