УДК 004.41+004.42 http://doi.org/10.15407/pp2024.02-03.239

К.П. Нестеренко, І.В. Стеценко

МЕТОД УПРАВЛІННЯ ВИКОНАННЯМ ЗАДАЧ БАГАТОПОТОКОВОЇ ПРОГРАМИ ЗА ЗАДАНИМ ГРАФОМ ЗАЛЕЖНОСТЕЙ

Швидкодія є однією з головних нефункціональних вимог до розробки програмного забезпечення. Як наслідок збільшення кількості ядер центральних процесорів протягом останніх десятиліть, використання технології багатопотоковості стало основним засобом підвищення швидкодії програмного забезпечення. У даному дослідженні аналізуються проблеми, які виникають в результаті розробки багатопотокових програм, та способи їх вирішення. Запропоновано метод управління виконанням задач багатопотокової програми за заданим графом залежностей та розглянута його реалізація мовою C++. Його метою є зменшення ресурсоємності розробки програмного забезпечення та збільшення його надійності шляхом вирішення проблем, характерних для багатопотокових програм. Наведено результати експериментального дослідження на тестовому наборі задач, що доводять збільшення швидкодії за рахунок використання запропонованого методу.

Ключові слова: програмне забезпечення, багатопотоковість, швидкодія, надійність, граф залежностей, паралельні обчислення, C++

K.P. Nesterenko, I.V. Stetsenko

METHOD OF MANAGING THE EXECUTION OF TASKS OF A MULTITHREADED PROGRAM ACCORDING TO A GIVEN DEPENDENCY GRAPH

Performance is one of the main non-functional requirements for software. As a result of the increase in the number of cores in central processing units in recent decades, the use of multithreading technology has become a primary means of improving software performance. This study analyzes the problems that arise from developing multithreaded programs and ways to address them. A method for managing the execution of tasks in a multithreaded program based on a given dependency graph is proposed and its implementation in the C++ language is demonstrated. Its aim is to reduce the resource intensity of software development and increase its reliability by addressing problems typical of developing multithreaded programs. The results of experimental research on a test set of tasks are provided, demonstrating increased performance through the use of the proposed method.

Keywords: software, multithreading, performance, reliability, dependency graph, parallel computing, C++

Вступ

Основна мета використання багатопотокового підходу під час розроблення програмного забезпечення – це підвищення показників швидкодії програмного коду. Проте інший, не менш важливий аспект багатопотокової розробки - це збереження надійності програмного забезпечення.

Використання багатопотоковості та інструментів синхронізації може призвести до ряду помилок: Data race, Race Condition, Deadlock, Livelock та Starvation [2].

Наявність цих проблем в коді під час виконання програми може призводити до різних наслідків - від крешів до непередбачуваної або некоректної поведінки програми. Підходи до вирішення проблем багатопотоковості можуть суттєво відрізнятися залежно від використовуваної мови програмування, фреймворку, засобів розробки тощо. Проте усі вони мають певні недоліки у застосуванні, оскільки можуть негативно впливати як на ресурсоємність процесу розробки, так і на швидкодію розробленого програмного коду. Принципово ці підходи можна поділити на два типи: ті, що не дозволяють виникнення помилки, та ті, що дозволяють інженеру

виявити помилку в тому випадку, коли вона сталася.

Метою даного дослідження є зменшення ресурсоємності розробки багатопотокових програм та підвищення їхньої надійності за рахунок розробки програмних засобів, які дають змогу запобігти виникненню або сприяти швидкому виявленню помилки в коді багатопотокової програми.

1. Огляд існуючих рішень

1.1. Data race. Data race виникає у тих випадках, коли два потоки звертаються одночасно до однієї і тієї самої ділянки пам'яті, водночас хоча б один з них виконує операцію запису [2]. Класичним механізмом для запобігання виникнення Data race у багатопотоковому коді є використання синхронізації. За допомогою синхронізованого доступу до даних, можливо гарантувати, що певна ділянка пам'яті одночасно може бути модифікована лише одним потоком, або що вона не буде модифікована в процесі вичитування її потоком. Інструменти синхронізації є невід'ємною складовою сучасних мов програмування і продовжують розвиватися, розширюючи можливості розробників у керуванні багатопоточною програмою та її ресурсами, наприклад, C++ Concurrency Library [3].

Проте використання примітивів синхронізації має свої недоліки, основним з яких є зменшення швидкодії програмного коду. Окрім того, що безпосередньо сам факт застосування примітиву, як-от, захоплення м'ютекса потоком, займає додатковий час на виконання, також значна кількість часу може бути втрачена потоками на очікування, доки звільниться ресурс, захищений примітивом синхронізації.

Для підвищення швидкодії програмного коду розробники все частіше застосовують підходи до розробки програмного забезпечення без використання примітивів синхронізації, які блокують виконання потоків [4]. Проте в такому випадку значно підвищується ризик виникнення проблеми Data race. Для вирішення цієї проблеми використовують різноманітні засоби, які виявляють дану помилку безпосередньо під час виконання програмного коду, наприклад, Data Race Detector у мові програмування Golang [5], або статично аналізуючи написаний програмний код [6].

Останнім часом з'являються напрацювання, пов'язані із застосуванням штучного інтелекту для виявлення Data race в програмному коді [7]. Проте використання подібних засобів для виявлення Data races теж має недоліки, оскільки вимагає впровадження додаткових інструментів у процесі розроблення та тестування програмного забезпечення, які збільшують загальну ресурсоємність розробки.

1.2. Race condition. Race condition, на відміну від Data race, є набагато складнішою для виявлення проблемою, оскільки природа її не технічна, а семантична. Визначення терміну Race condition можна формалізувати як проблему, яка виникає внаслідок того, що від порядку виконання операцій в потоках, змінюється результат виконання програми [8].

Аналогічно з помилкою Data race, для виявлення Race condition існують певні засоби, які за допомогою статичного аналізу коду [9] або за рахунок виявлення небезпечних шаблонів під час виконання коду [10] попереджають розробника про потенційну можливість виникнення даної проблеми. Проте точність подібного виявлення часто не є задовільною.

Водночас формалізованих методів та засобів для уникнення Race condition в загальному випадку немає. Зменшити кількість даних помилок або уникнути їх в цілому можна тільки використовуючи певні методи та архітектурні підходи до розробки багатопотокових програм [11].

1.3. Deadlock. Deadlock – це стан багатопотокової програми, коли два або більше потоків не можуть продовжити своє виконання через взаємне блокування одне одного. Класичною стратегією для уникнення проблеми Deadlock є підхід, в якому отримання потоком усіх необхідних йому ресурсів відбувається виключно на початку його виконання [12]. Таким чином унеможливлюється ситуація, коли може утворитися циклічна залежність між двома потоками. Потік або виконується, або чекає доки не звільняться усі потрібні йому

ресурси. Проте даний підхід негативно впливає на показники швидкодії, оскільки в загальному випадку потоку не потрібні усі ресурси в кожен окремий момент виконання, і відповідно в цей час вони могли б бути використані іншими потоками.

Для виявлення Deadlock існує багато різноманітних методів та алгоритмів. Наприклад, можна за допомогою таймерів контролювати час виконання певних критичних ділянок коду у потоках і, якщо цей час виконання перевищує певне граничне значення, повідомляти розробника про потенційний Deadlock [12]. Також варто зазначити, що для Deadlock існують стратегії відновлення виконання програми у випадку його виникнення [13], що надзвичайно важливо для систем з підвищеними вимогами до надійності.

1.4. Starvation. Starvation – це стан багатопотокової програми, коли один з потоків не може виконувати задачу через неможливість отримати доступ до потрібних ресурсів. Такий стан може виникнути через помилки в проєктуванні багатопотокових програм, коли захищені примітивами синхронізації ресурси використовуються потоками впродовж тривалого часу. Через це інші потоки витрачають значну кількість часу в очікуванні звільнення потрібного ресурсу.

Для уникнення Starvation зазвичай використовують алгоритми пріоритезації [14] або планування задач [15], які гарантують, що в певний момент часу задача буде виконана, та дозволяють розподілити задачі таким чином, щоб мінімізувати витрати часу на очікування звільнення ресурсів.

Для виявлення Starvation можуть використовуватися різноманітні програмні засоби, які досліджують метрики про час виконання окремих функцій в потоках і дають змогу розробнику виявити та виправити проблемні ділянки програмного коду [16]. Подібні засоби часто інтегрують в процеси автоматизації тестування програмного забезпечення для того, щоб постійно контролювати стан програмного коду та повідомляти розробників про аномалії чи проблеми щодо швидкодії.

В результаті аналізу існуючих проблем багатопотоковості та можливих варіантів їх рішень, можна стверджувати, що вирішення даних проблем суттєво збільшує ресурсоємність розробки багатопотокового програмного забезпечення. Також деякі з існуючих рішень негативно впливають на швидкодію програмного забезпечення, що ставить розробника перед складним вибором між швидкодією та надійністю програмного продукту.

Отже, проблема зменшення ресурсоємності розробки багатопотокового програмного забезпечення без втрат його надійності та показників швидкодії, вимагає подальшого дослідження.

2. Метод управління виконанням задач багатопотокової програми за заданим графом залежностей

2.1. Загальний опис методу. Основна ідея методу управління виконанням задач багатопотокової програми на основі графу залежностей полягає в тому, щоб задати процес виконання задач, на які розбиті обчислення, у вигляді графу залежностей, де кожна вершина графа відповідає певній конкретній задачі, а дуги графу позначають залежності між цими задачами. Водночас задачі можуть бути виконані паралельно в потоках з урахуванням обмежень, накладених залежностями між ними.

Метод управління виконанням задач багатопотокової програми на основі графу залежностей дає змогу описати процес виконання багатопотокового програмного коду у зрозумілій та добре структурованій формі, а також уникати або виявляти проблеми, що виникають в результаті використання синхронізації виконання задач.

2.2. Формалізований опис методу. Припустимо, що для успішного виконання програми вона має виконати K задач : $A=\{A_0, \ldots, A_{K-1}\}$. Кожній задачі з множини A ставиться у відповідність вершина графу G: $G_0, \ldots, G_{K-1}$. Якщо між задачами A_i та A_j, де $0 \le i, j < K$, існує семантична залежність або задачі A_i та A_j під час свого виконання використовують спільний ресурс, то між відповідними вершинами

графу G_i та G_j існує зв'язок і йому у відповідність ставиться у графі дуга (G_i, G_j). Під семантичною залежністю задач розуміємо таку їх залежність, коли результат виконання програми залежить від послідовності виконання задач A_i та A_j і водночас тільки один з можливих результатів є коректним.

Задачі в ході виконання програми можуть дозавантажуватись, утворюючи нові вершини та нові дуги у графі, та видалятись у разі успішного виконання. Задача A_x може бути виконана в програмі тоді і тільки тоді, коли у графі залежностей не існує жодної дуги (G_i, G_x), $0 \le i < K$. Якщо задача A_x виконана, вона видаляється з графу разом з дугами (G_x, G_i), де $0 \le i < K$. Виконання програми вважається успішним, коли множина вершин графу залежностей G на момент завершення програми є порожньою, тобто усі завантажені на виконання задачі були виконані.

2.3. Програмний опис методу. Основою реалізації методу управління виконанням задач багатопотокової програми на основі графу залежностей є імплементація класу GraphManager, що буде відповідати за порядок виконання задач. Відповідно даний клас повинен контролювати список наявних задач, обирати задачі, які можуть бути виконані, та після їх виконання оновлювати існуючі залежності.

Для представлення задачі у вигляді вершини графу залежностей потрібно створити клас GraphNode, який буде містити інформацію про задачу, яка має бути виконана, та список задач, від яких залежить вона та залежні від неї. Залежності між задачами встановлюються розробником після того, як відповідні об'єкти класу GraphNode були додані у GraphManager.

Під час імплементації методу розробник повинен представити наявні в програмі типи задач у вигляді окремих класів SomeConcreteJob, які є нащадками абстрактного класу AbstractJob.

2.4. Вирішення проблем багатопотоковості за допомогою методу. Для уникнення проблеми Race condition розробнику достатньо вказати на залежність між задачами, від порядку виконання яких залежить коректність результату роботи програми. Метод гарантує, що задачі, зі встановленими залежностями, будуть виконані саме у вказаному порядку одна відносно одної.

Для уникнення проблеми Data race та покращення показників швидкодії, за рахунок можливості частково відмовитися від примітивів синхронізації для захисту доступу до певних об'єктів, розробник має можливість встановити залежність між задачами, які вимагають ексклюзивний доступ до певних ресурсів на час свого виконання, та дозволити паралельно виконувати задачі, які не модифікують спільні ресурси.

Не зважаючи на те, що запропонований метод не дозволяє уникнути виникненню проблем Deadlock та Livelock, з його допомогою досить просто імплементувати механізм, який буде виявляти дані проблеми ще до початку виконання графу залежностей. Оскільки проблема Deadlock виникає у випадку наявності взаємної залежності між синхронізованими ресурсами у двох потоках, то в контексті даного методу, задача пошуку потенційних Deadlock зводиться до задачі пошуку циклів у графі, для вирішення якої існує багато розроблених алгоритмів [17].

Запропонований метод не дозволяє повністю уникнути проблеми Starvation. Проте він дозволяє її мінімізувати, а також імплементувати механізми моніторингу за часом виконання та очікування задач без використання сторонніх засобів контролю. Метод управління виконанням задач багатопотокової програми на основі графу залежностей дає змогу розробнику налаштувати граф таким чином, щоб жодна задача не мала очікувати на звільнення певного спільного ресурсу після початку свого виконання. Натомість така задача просто не буде запущена допоки ресурс не звільниться, тобто не буде виконана вимога залежності між задачами. Відповідно в цей час ресурси системи можуть бути спрямовані на виконання задачі, яка в даний момент часу не має залежностей, або залежності якої вже були виконані. Проте, це не дозволяє уникнути випадку, коли існує задача, на яку існує залежність у великої кількості інших задач, а відповідно вони не

можуть виконуватися допоки вона успішно не завершиться. Для виявлення подібних проблем нескладно імплементувати механізми, що будуть слідкувати за станом графу, фіксувати для кожної задачі час її виконання та час її очікування на виконання тощо. На основі цієї інформації розробник зможе вносити зміни до програмного коду, націлені на виправлення проблених ділянок коду та покращення показників швидкодії програмного засобу.

3. Експериментальне дослідження ефективності методу управління виконанням задач багатопотокової програми за заданим графом залежностей

3.1. Опис тестового програмного забезпечення. Для перевірки ефективності запропонованого методу, була розроблена програма мовою C++, яка за допомогою шаблону проєктування Strategy, дозволяє виконати один і той самий набір задач або за допомогою звичайної реалізації багатопотокової програми, або на основі графу залежностей. Для управління потоками використовується шаблон проєктування Thread Pool (рис. 1).

Для того, щоб мати можливість конфігурувати, за допомогою якого підходу виконувати задачі, був створений інтерфейс IJobManager (рис. 2). Інтерфейс інкапсулює механізм, за допомогою якого Thread Pool обирає наступну задачу для виконання.

```
#pragma once
#include <vector>
#include <thread>

class IJobManager;

class ThreadPool
{
public:
    ThreadPool(IJobManager* pJobManager,
        size_t numThreads = std::thread::hardware_concurrency());
    ~ThreadPool();

    void Start();
    void Stop();

    void ThreadMainLoop();

    IJobManager* GetJobManager() { return m_jobManager; }

private:
    size_t m_numThreads { 0 };

    IJobManager* m_jobManager { nullptr };
    std::vector<std::thread> m_threads;
};
```

Рис. 1. Імплементація шаблону проєктування ThreadPool у вигляді класу.

```
#pragma once

#include<memory>

class BaseJob;

class IJobManager
{
public:
    virtual void AddJob(BaseJob* pJob) = 0;
    virtual std::unique_ptr<BaseJob> GetNextJob() = 0;

    virtual void OnJobFinished(int jobId) = 0;

    virtual bool IsFinished() = 0;
};
```

Рис. 2. Клас IJobManager.

Інтерфейс IJobManager імплементований у класах DefaultJobManager (рис. 3) та GraphJobManager (рис. 4). Клас DefaultJobManager імплементований за допомогою класичної черги за принципом «перший зайшов – перший вийшов». Відповідно він відображає класичний сценарій написання багатопотокової програми, де відповідальність за синхронізацію повністю покладена на примітиви синхронізації.

```
#pragma once
#include "IJobManager.h"
#include <mutex>
#include <queue>

class DefaultJobManager
    : public IJobManager
{
public:
    virtual void AddJob(BaseJob* pJob) override;
    virtual std::unique_ptr<BaseJob> GetNextJob() override;

    virtual void OnJobFinished(int jobId) override {}

    virtual bool IsFinished() override;

private:
    mutable std::mutex m_jobQueueMutex;
    std::queue<std::unique_ptr<BaseJob>> m_jobQueue;
};
```

Рис. 3. Клас DefaultJobManager.

GraphJobManager в свою чергу імплементований за допомогою запропонованого методу. В середині класу міститься структура даних m_graphMap, яка ставить кожній задачі у відповідність вершину графу, описану за допомогою класу GraphNode (рис. 5).

```
#pragma once
#include "IJobManager.h"
#include <mutex>
#include <queue>
#include <map>

#include "GraphNode.h"

class BaseJob;

class GraphJobManager :
    public IJobManager
{
public:
    virtual void AddJob(BaseJob* pJob) override;
    virtual std::unique_ptr<BaseJob> GetNextJob() override;

    virtual void OnJobFinished(int jobId) override;

    virtual bool IsFinished() override;

    void AddJobDependency(int jobId, int dependentJobId);

private:
    int m_nextJobId { 0 };

    mutable std::mutex m_jobQueueMutex;
    std::queue<std::unique_ptr<BaseJob>> m_jobQueue;

    std::map<int, std::unique_ptr<GraphNode>> m_graphMap;
};
```

Рис. 4. Клас GraphJobManager.

```
#pragma once
#include <memory>
#include <vector>
#include <atomic>

class BaseJob;

class GraphNode
{
public:
    GraphNode(BaseJob* job);

    void AddDependency(GraphNode* node);
    void ResolveDependencies();

    void IncUnresolvedDependencies();
    void DecUnresolvedDependencies();

    bool CanExecuteJob();
    std::unique_ptr<BaseJob> ExtractJob();

private:
    std::unique_ptr<BaseJob> m_job;
    std::atomic_int m_unresolvedDependencies { 0 };
    std::vector<GraphNode*> m_dependentNodes;

    bool m_bJobExtracted { false };
};
```

Рис. 5. Клас GraphNode.

Кожен екземпляр класу GraphNode зберігає в середині себе кількість невиконаних залежностей для відповідної задачі, або іншими словами, кількість дуг графу, які входять у дану його вершину. Також кожен екземпляр зберігає покажчики на інші екземпляри класу GraphNode, фактично описуючи дугу графу що виходять з даної вершини.

Коли для вершини графу виконані усі залежності, вона переміщується у чергу для виконання в класі GraphJobManager. Також в класі GraphJobManager присутня функція AddJobDependency, яка дозволяє вказати на залежність між задачами, тобто фактично створити нову дугу у графі залежностей.

3.2. Опис тестового набору даних та сценаріїв тестування. Для тестування Thread Pool був сконфігурований пул 4 потоків. Набір тестових задач складений з 4 груп по 100 задач, які можна представити у вигляді множин $A=\{A_0, \ldots, A_{99}\}$, $B=\{B_0, \ldots, B_{99}\}$, $C=\{C_0, \ldots, C_{99}\}$, $D=\{D_0, \ldots, D_{99}\}$. Всередині кожної групи, імітуючи реальні умови виконання для багатопотокової задачі, вважаємо, що існує певний ресурс, до якого на час виконання задача отримати ексклюзивний доступ за допомогою м'ютекса. Після цього задача 10 мс імітує виконання обчислень, використовуючи підхід Busy wait [18], і успішно завершує своє виконання. Використання саме Busy wait обумовлено прагненням уникнути впливу механізмів оптимізації використання потоків, реалізованих в операційній системі, на отриманий результат.

Для отримання більш точних результатів для порівняння тестовий набір задач виконується 100 разів для кожної з реалізацій. При цьому збираються наступні метрики: найшвидший час виконання, найгірший час виконання, середній час виконання.

Щоб отримати показники часу виконання програми, для обчислення заданого набору задач за допомогою графу залежностей, побудованого на основі запропонованого методу, замість DefaultJobManager буде використаний GraphJobManger. Додаючи задачі у GraphJobManger, їй у відповідність буде створена вершина графу, описана за допомогою класу GraphNode. Далі за допомогою функції AddJobDependency будуть створені залежності між задачами, тобто додані дуги у графі залежностей. В даному випадку об’єктом залежностей є ресурс, виділений для кожної з груп задач. Тоді за допомогою функції AddJobDependency створимо залежності між задачами наступним чином: кожна задача A_i буде залежати від виконання задачі A_{i-1}, B_i від B_{i-1}, C_i від C_{i-1} та , D_i від D_{i-1} де $i \in [1, 99]$.

Варто зазначити, що, оскільки класичний підхід до виконання багатопотокових задач імплементований за допомогою черги, загальний час виконання задач буде залежати від їх порядку в цій черзі. Відповідно було розроблено 3 сценарії для тестування класичного підходу: найкращий, найгріший та реалістичний.

Найкращим сценарієм буде вважатися випадок, коли задачі у черзі розміщені так, що кількість часу, витрачена задачами на очікування на доступ до ресурсів, є мінімальною. Такий сценарій буде відповідати розміщенню задач у черзі, коли вона складається з послідовностей задач, кожна з яких знаходиться у різній групі, а відповідно вимагає різний ресурс для свого виконання. Тобто послідовність задач в черзі матиме наступний вигляд: {A_0, B_0, C_0, D_0, …, A_{99}, B_{99}, C_{99}, D_{99}}

Найгіршим сценарієм буде вважатися випадок, коли задачі у черзі розміщені таким чином, що кількість часу, витрачена задачами на очікування на доступ до ресурсів, є максимальною. Такий сценарій буде відповідати розміщенню задач у черзі, коли групи задач (див. підрозділ 3.2) розміщені у черзі послідовно. Тобто послідовність задач в черзі матиме наступний вигляд: {A_0, …, A_{99}, B_0, …, B_{99}, C_0, …, C_{99}, D_0, …, D_{99}}.

Реалістичним буде вважатися сценарій, де задачі розміщуються у черзі випадковим чином, імітуючи умови наближені до тих що можуть статися під час реального виконання програми.

3.3. Результати тестування. Результати експериментів, представлені у таблиці 1, свідчать, що у порівнянні з реалістичним та найгіршим сценаріями виконання, запропонований метод продемонстрував значно вищу швидкодію. Цей результат було досягнуто за рахунок того, що дозволяє розробнику самостійно задати порядок виконання задач за допомогою введення залежностей між ними (див. підрозділ 2.1), тим самим зменшивши вплив механізмів синхронізації на загальний час виконання програми.

Результат у найкращому сценарії виконання демонструє мінімальну різницю у швидкодії на користь класичного підходу. Це легко пояснюється тим, що у даному випадку порядок виконання задач в обох підходах співпадає, проте запропонований метод має додаткову витрату часу на побудову та оновлення графу залежностей. Проте, враховуючи, що в реальних умовах подібний сценарій може трапитися надзвичайно рідко.

Таблиця 1

Результати експериментального дослідження швидкодії виконання задач

Сценарій виконання задач	Час виконання, мс		
	найменший	найбільший	середній
найкращий	1002,95	1007,52	1003,50
найгірший	3913,40	3918,90	3914,39
реалістичний	1513,09	1603,00	1548,52
за графом залежностей	1005,07	1019,02	1006,70

Отже, за результатами експериментального дослідження доведено, що запропонований метод дає змогу підвищити швидкодію тестової програми в середньому на 35% у порівнянні з класичним підходом: $\frac{(1548,52-1006,7)}{1548,52} \cdot 100 = 35\%$.

Висновки

Проведено аналіз проблем, які виникають під час розробки багатопотокових програм. Для кожної з проблем були наведені актуальні методи їхнього вирішення, визначені переваги та недоліки.

Запропоновано метод управління виконанням задач багатопотокової програми за заданим графом залежностей. Описано загальну ідею методу, формальний опис методу, програмний опис методу, а також, яким чином даний метод вирішує зазначені проблеми і які переваги для розробника він забезпечує у порівнянні з існуючими методами.

Розроблено тестове програмне забезпечення та тестовий набір задач, на якому проведено заміри швидкодії запропонованого методу та класичного підходу до виконання задач у потоках. Проаналізувавши отриманий результат, було зроблено висновок, що запропонований метод дає змогу підвищити швидкодію тестової програми в середньому на 35% у порівнянні з класичним підходом.

В подальшому планується продовжити розвиток використання методу для розробки багатопотокових програм на основі графу залежностей за рахунок створення повноцінного середовища розробки багатопотокових програм на його основі.

Література

1. S. Borkar, and Chien, A. (2011) 'The Future of Microprocessors', *Communications of the ACM*, 54, 67-77.
2. Yuan L., (2006) 'Multithreaded programming challenges, current practice, and languages/tools support', in *2006 IEEE Hot Chips 18 Symposium (HCS)*, Stanford, CA, 2006, pp. 1-134.
3. Gregoire, M. (2021) 'Multithreaded Programming with C++', in *Professional C++*, 5th Edition. John Wiley & Sons, pp. 915-967.
4. Fraser, K. and Harris, T. (2007) 'Concurrent programming without locks', *ACM Transactions on Computer Systems*, 25, 2, 5–es. Available at: https://doi.org/10.1145/1233307.1233309 (Accessed: 13 April 2024).
5. Chabbi, M. and Ramanathan, M. (2022) 'A study of real-world data races in Golang', in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*, pp. 474-489.
6. Kahlon, V. *et el.* (2007), 'Fast and Accurate Static Data Race Detection for Concurrent Programs', *Lecture Notes in Computer Science*, 4590, 226-239.
7. Chen, L. *et el.* (2023). Data Race Detection Using Large Language Models' in *SC-W '23: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pp. 215–223.
8. Netzer, R. and Miller, B. (1992) 'What are Race Conditions?: Some Issues and Formalizations', *ACM letters on programming languages and systems*, 1, 74-88. Available at: https://doi.org/10.1145/130616.130623 (Accessed: 13 April 2024).
9. Flanagan, C. and Freund, S. (2001). Detecting Race Conditions in Large Programs' in *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 90-96.
10. Yousaf, M. *et el.* (2021) 'Efficient Identification of Race Condition Vulnerability in C Code by Abstract Interpretation and Value Analysis' in *IEEE International Conference on Cyber Warfare and Security*. Available at: https://doi.org/10.1109/ICCWS53234.2021.9702954 (Accessed: 13 April 2024)
11. Ortega-Arjona, J.L. (2004) 'The Manager Workers Pattern' in *European Conference on Pattern Languages of Programs*, pp. 53-64.
12. Singhal, M. (1989) 'Deadlock detection in distributed systems', *Computer*, 22, 37 - 48.
13. Park, Y., Scheuermann, P. and Tung, H. L. (1995), 'A Distributed Deadlock Detection and Resolution Algorithm Based on A Hybrid Wait-for Graph and Probe Generation Scheme', in *CIKM '95: Proceedings of the fourth international conference on Information and knowledge management*, pp. 378-386.
14. Jabbour, R. and Elhajj, I. (2008) 'SAF-PS: Starvation Avoidance for Priority Scheduling' in *2008 5th International Multi-Conference on Systems, Signals and Devices*, SSD'08, 1-6. Available at: https://doi.org/10.1109/SSD.2008.4632789 (Accessed: 13 April 2024).
15. Gawanmeh, A. *et el.* (2021). Starvation Avoidance Task Scheduling Algorithm for Heterogeneous Computing Systems. Available at: https://doi.org/10.1109/CSCI54926.2021.00339 (Accessed: 13 April 2024).
16. Abbaspour, S. *et el.* (2016) 'A Model for Systematic Monitoring and Debugging of Starvation Bugs in Multicore Software'. Available at: https://doi.org/10.1145/2975954.2975958 (Accessed: 13 April 2024).
17. Nesterenko, A. (2012) 'Cycle detection algorithms and their applications', *Journal of Mathematical Sciences*, 182, 518-526.
18. Blieberger, J., Burgstaller, B., Scholz, B. (2003), 'Busy Wait Analysis', *Lecture Notes in Computer Science*, 2655, 142-152.

Одержано: 06.04.2024
Внутрішня рецензія отримана: 22.04.2024
Зовнішня рецензія отримана: 27.04.2024

***Про авторів*:**

[1]*Нестеренко Костянтин Павлович,*
аспірант кафедри інформатики та програмної інженері
https://orcid.org/0000-0003-3921-4324

[1]*Стеценко Інна Вячеславівна*,
доктор технічних наук, професор,
професор кафедри інформатики
та програмної інженерії
http://orcid.org/ 0000-0002-4601-0058

Місце роботи авторів:
[1]Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»,
03056, м. Київ,
проспект Берестейський 37.
Тел.: (044) 236-9651
e-mail: k.nesterenko@kpi.ua,
stiv.inna@gmail.com