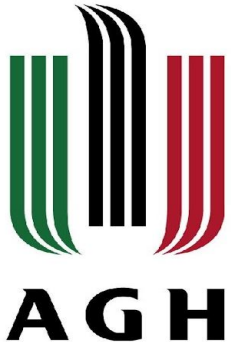


AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE
AGH UNIVERSITY OF SCIENCE
AND TECHNOLOGY

Thread pool - zarządzanie pulą wątków

Filip Zajdel
19.01.2022

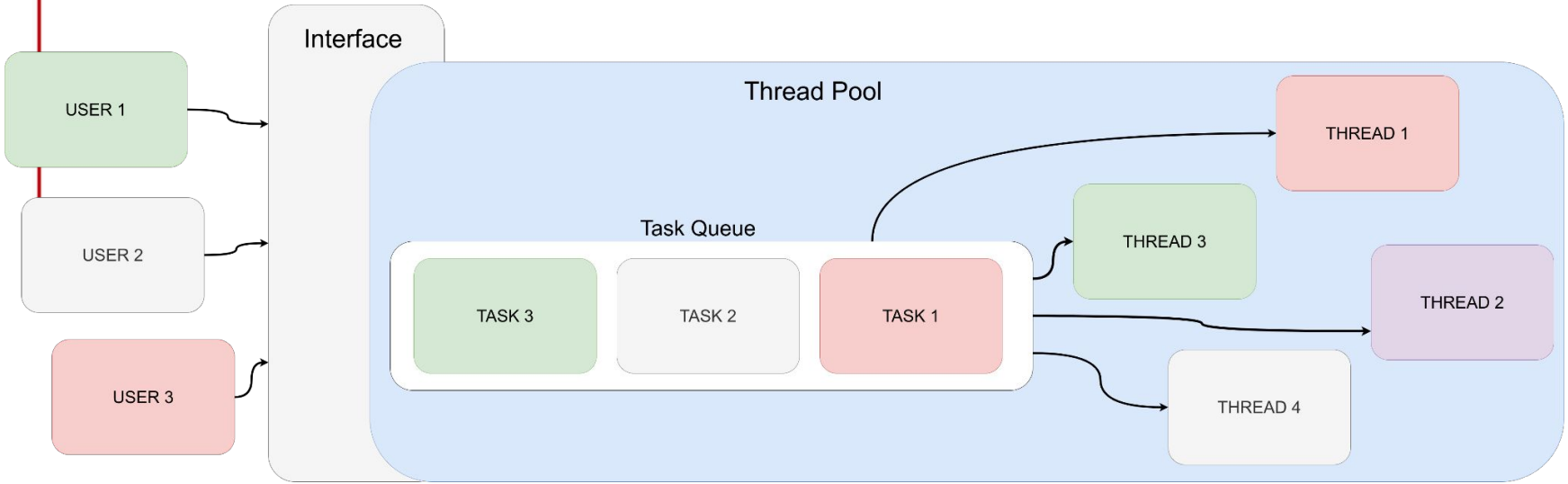
Problemy z wykonywaniem niewielkich zadań w wielu wątkach

Oddelegowywanie pomniejszych zadań do osobnych wątków może prowadzić do:

- nieefektywnie wysokiego **zużycia procesora** na tworzenie, destrukcję i przełączanie pomiędzy wieloma wątkami, względem czasu poświęconego na egzekucję samych wątków
- nadmiernego **zużycia pamięci** zaalokowanej dla poszczególnych wątków
- **niekontrolowanej liczby** równocześnie działających wątków- wpływa to na stabilność systemu

Thread pool - pula wątków

Thread pool jest wzorcem, który **zarządza** pulą wątków i **przydziela** im otrzymane z zewnątrz zadania (taski).



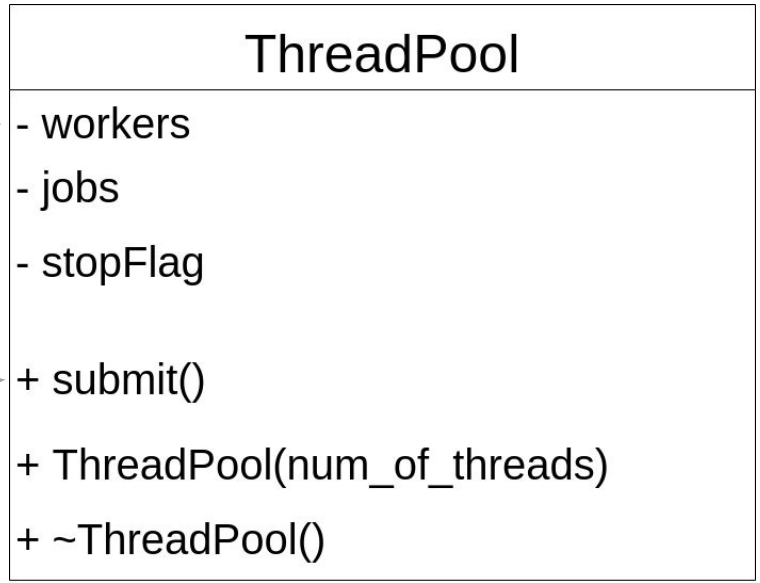
Podstawowe funkcjonalności puli wątków

1. Konstrukcja określonej liczby wątków (np. maksymalnej liczby wspieranej przez hardware lub określonej przez użytkownika).
2. Gromadzenie zadań w określonej strukturze danych (np. kolejka) i umożliwienie wątkom pobierania zadań.
3. Udostępnienie interfejsu użytkownika do:
    - oddelegowywania zadań do wykonania przez thread pool
    - zakończenia funkcjonowania puli wątków

Podstawowe funkcjonalności puli wątków

przykład implementacji

1. Konstrukcja określonej liczby wątków (np. maksymalnej liczby wspieranej przez hardware lub określonej przez użytkownika).
2. Gromadzenie zadań w określonej strukturze danych (np. kolejka) i umożliwienie wątkom pobierania zadań.
3. Udostępnienie interfejsu użytkownika do:
   - oddelegowywania zadań do wykonania przez thread pool
   - zakończenia funkcjonowania puli wątków



Podstawowe funkcjonalności puli wątków

przykładowa implementacja w C++

```
class ThreadPool {
    typedef function<void(void)> job_t;
    queue<job_t> jobs;
    vector<thread> workers;
    mutex jobs_mutex;
    atomic_bool stopFlag;
    static void worker(ThreadPool *pool);

  public:
    ThreadPool(unsigned num_of_threads);
    ~ThreadPool();
    void submit(job_t job);
};
```

Podstawowe funkcjonalności puli wątków

Implementacja cd.

```
void ThreadPool::worker(ThreadPool *pool)
{
    while(!pool->stopFlag) {
        job_t job;
        bool got_job = false;
        {
            lock_guard<mutex> lock(pool->jobs_mutex);
            if (!pool->jobs.empty()) {
                job = pool->jobs.front();
                pool->jobs.pop();
                got_job = true;
            }
        }
        if (got_job) {
            // cout << "Executing job on thread " << s
            job();
        }
    }
}
```

Podstawowe funkcjonalności puli wątków

Implementacja cd.

```
ThreadPool::ThreadPool(unsigned num_of_threads) : stopFlag(false)
{
    for (unsigned i=0; i<num_of_threads; i++) {
        workers.push_back(thread(ThreadPool::worker, this));
    }
}

void ThreadPool::submit(ThreadPool::job_t job)
{
    lock_guard<mutex> lock(jobs_mutex);
    jobs.push(job);
}

ThreadPool::~ThreadPool()
{
    stopFlag = true;
    for (auto &worker: workers) {
        worker.join();
    }
}
```

Podstawowe funkcjonalności puli wątków

demo

Porównanie czasu wykonania następujących funkcji.

```
void execute_with_many_threads(unsigned num_of_jobs)
{
    vector<thread> threads;
    threads.reserve(num_of_jobs);

    for (int i=0; i<num_of_jobs; i++) {
        threads.push_back(
            thread([](){
                for(unsigned j=1; j<=100000; j++) {
                }
            }));
    }

    for (auto &thread: threads)
    {
        thread.join();
    }
}
```

Podstawowe funkcjonalności puli wątków

demo cd.

Porównanie czasu wykonania następujących funkcji.

```
void execute_with_thread_pool(unsigned num_of_jobs)
{
    ThreadPool threadPool(thread::hardware_concurrency());

    for (int i=0; i<num_of_jobs; i++) {
        threadPool.submit([](){
            for(unsigned j=1; j<=100000; j++) {
            }
        });
    }
}
```

Kiedy użycie thread pool jest korzystne?

- Wykonanie kolejnego zadania wymaga kosztownej alokacji nowych zasobów (np. otwarcie socketu). Thread może wówczas współdzielić dane zasoby pomiędzy wątkami.
- Zadania do wykonania są od siebie niezależne.
- Wykonywane zadania nie są blokujące.
- Łatwo jest podzielić dane zadanie na mniejsze podzadania do wykonania przez thread pool.

Przykładowe biblioteki implementujące thread pool

- OpenMP
- Boost
- QThreadPool