1. WORK POOL

The work pool pattern (also known as the worker pool or thread pool pattern) in C++ is a common concurrency design where a fixed number of worker threads pull tasks from a shared queue and process them concurrently. Work pool is used to efficiently utilize system resources, especially when we have many short-lived tasks. A typical implementation of this parallel pattern includes:

- **Task Queue:** A thread-safe queue holding tasks.
- **Worker Threads:** A pool of threads that keep pulling and executing tasks.
- **Synchronization:** Mechanisms that ensure thread-safe access to the queue.
- **Graceful Shutdown:** Mechanism to stop workers cleanly.

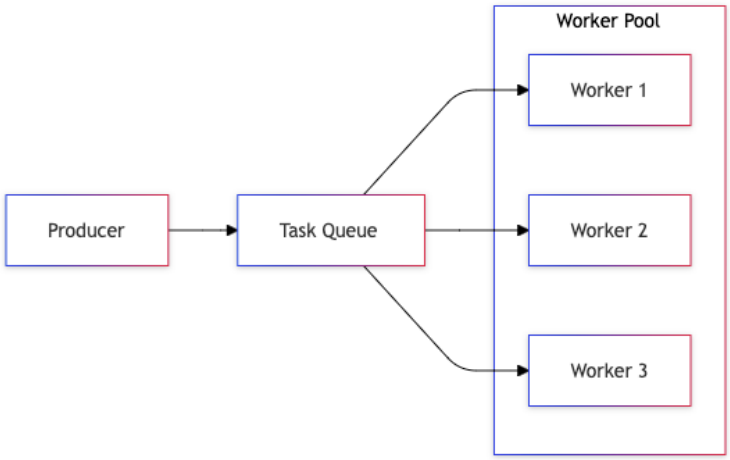

Figure 1: Conceptual design of work pool pattern (Source: DZone).

In general, producers create tasks and submit them to a shared queue which serves as a buffer between the producers and the worker threads, holding all pending tasks waiting to be processed. In contrast, worker threads constantly check the queue, and when new tasks exist, they take it from the queue and run independently. The workers then goes back to waiting for the next one after completing the current task. This process repeats continuously until the thread pool is shut down.

Since multiple producers and worker threads access the task queue concurrently, synchronization is necessary to prevent race conditions. For simplicity, this course only requires students to use mutex locks, however, implementing thread-safe data structures such as lock-free queues is highly encouraged. The following code shows a conceptual implementation of work pool model.

```
#include <vector>
#include <thread>
#include <queue>
```

```
#include <mutex>
#include <stdio.h>

#define NUM_THREAD 4
#define NUM_TASK 1000

struct Task {
    int id;
    Task(int id) : id(id) {}
};

class TaskQueue {
    private:
        std::mutex mtx;
        std::queue<Task*> queue;
        static TaskQueue* instance;

    public:
        void enqueue(Task* task) {
            std::lock_guard<std::mutex> lock(mtx);
            queue.push(task);
        }

        Task* dequeue() {
            std::lock_guard<std::mutex> lock(mtx);
            if (!queue.empty()) {
                Task* task = queue.front();
                queue.pop();
                return task;
            }
            return nullptr;
        }

        static TaskQueue* get() {
            if (instance == nullptr)
                instance = new TaskQueue();
            return instance;
        }
};

class Worker {
    private:
        bool stop;
        std::thread t;
        int id;

        void run() {
            while (!stop) {
                Task* task = TaskQueue::get()->dequeue();
                if (task != nullptr) {
                    printf("Worker %d executes task %d\n", this->id, task->id);
                    delete task;
                } else {
                    // Sleep briefly to avoid busy-waiting (optional)
                    std::this_thread::sleep_for(std::chrono::milliseconds(10));
                }
            }
        }

    public:
        Worker(int id) : id(id), stop(false) {
            t = std::thread(&Worker::run, this);
        }

        void exit() {
            stop = true;
            t.join();
        }
};

TaskQueue* TaskQueue::instance = nullptr;

int main() {
    std::vector<Worker*> workers;

    for (int i = 0; i < NUM_THREAD; ++i)
        workers.push_back(new Worker(i));

    for (int i = 0; i < NUM_TASK; ++i)
        TaskQueue::get()->enqueue(new Task(i));

    // Allow workers some time to process tasks
    std::this_thread::sleep_for(std::chrono::seconds(1));

    // Shutdown all workers
```

```
    for (Worker* worker : workers) {
        worker->exit();
        delete worker;
    }

    return 0;
}
```

2. SHOCK WAVE BLAST

In a nuclear explosion, the surrounding air is violently compressed, creating a shock wave blast that travels outward. This wave is characterized by a sudden rise in pressure, called overpressure. This overpressure is calculated through the Scaled Distance (Hopkinson-Cranz Scaling Law) depicted as below:

$$\boxed{Z = R * W^{-1/3}}$$

- **Z =** scale distance $(m/kg^{1/3})$.
- **R =** distance from the blast (m).
- **W =** explosive yield (TNT equivalent, in kg).

Peak overpressure (in kPa) is the maximum pressure above normal atmosphere and directly relates to the damage potential. This metrics is mainly conducted from empirical estimations (Kingery-Bulmash data). To estimate the peak overpressure metric, we first calculate the intermediate value u for ease of later computation:

$$\boxed{U = -0.21436 + 1.35034 * \log_{10}(Z)}$$

The peak overpressure metric is then calculated indirectly through the following formula below, where coefficient c_i is conducted through experiments:

$$\boxed{\log_{10}(P_{so}) = \sum_{i=0}^{i=8} C_i * U^i}$$

- $c_0 = 2.611369.$
- $c_1 = -1.690128.$
- $c_2 = 0.00805.$
- $c_3 = 0.336743.$
- $c_4 = -0.005162.$
- $c_5 = -0.080923.$
- $c_6 = -0.004785.$
- $c_7 = 0.007930.$
- $c_8 = 0.000768.$

In reality, shock waves initially travel faster than the speed of sound due to their high overpressure, gradually slowing down as they expand outward. However, for a simple estimation, the arrival time of a shock wave can be approximated by assuming that it travels at the speed of sound. In particular, the time cost to reach distance R is:

$$\boxed{t = \frac{R}{343}}$$

3. EXERCISES

Assume all DF-61 warheads is detonated at the center of a map represented by a 4000×4000 matrix initially filled with zeros; each cell represents a 10 meters by 10 meters area. GPT expert estimates that the explosion has an approximate yield of total 5000 kilotons. In this exercise, students must model the blast shock wave for the first 100 seconds. The simulation advances in one-second time steps; when the shock front reaches a cell, replace that cell value with the corresponding peak overpressure at that location. In particular, students are required to:

- Develop a sequential (single-threaded) implementation of the simulation.
- Implement a parallel version of the simulation using the work pool model.
- Conduct a time comparison between the sequential and parallel implementations.

**** Submission:** Works must be done individually.

- Compress all necessary files as **"Lab_3_<Student ID>.zip"**.
- Students must submit and demonstrate their results before the end of the class.