GosFrontline笔记

So this is what I plan to do with a Renju game:

1. I will use three threads to monitor input (either from cli or from gui), perform calculations, and another one to output things to the interface. How is this usually implemented? Do I use independent threads or do I use a thread pool? How do I manage and save the status of these threads?
2. About intrathread communication: How can this be implemented? Must I use a global variable, or should I use pointers that point to the same location in the memory? E.g., when the user clicks takeback or resign, the calculations on the thread that takes charge of communications should immediately stop. How can this be done?
3. What about a multithread in multithread? Ie, will using a thread pool to do the search algorithm make my program go faster?

Planning out a Renju game with multi-threading is an exciting and complex task! Let's break down your questions and explore some potential solutions:

**1. Implementation of Threads**

* **Independent Threads vs. Thread Pool**:
  + **Independent Threads**: You can create separate threads for each specific task, like input monitoring, calculations, and output. This approach is straightforward but may lack scalability.
  + **Thread Pool**: A thread pool can be more efficient if you have multiple tasks to handle. It allows for reusing threads, reducing the overhead of thread creation and destruction.
  + **Management and Status**:
    - Use thread-safe data structures and synchronization primitives (e.g., std::mutex, std::condition\_variable) to manage thread communication and status.
    - Consider using a task scheduler within a thread pool to manage the workload dynamically.

**Example with Thread Pool:**

cpp

#include <iostream>

#include <thread>

#include <vector>

#include <queue>

#include <functional>

#include <mutex>

#include <condition\_variable>

class ThreadPool {

public:

ThreadPool(size\_t numThreads);

~ThreadPool();

void enqueueTask(std::function<void()> task);

private:

std::vector<std::thread> workers;

std::queue<std::function<void()>> tasks;

std::mutex tasksMutex;

std::condition\_variable condition;

bool stop;

void worker();

};

ThreadPool::ThreadPool(size\_t numThreads) : stop(false) {

for (size\_t i = 0; i < numThreads; ++i) {

workers.emplace\_back([this] { this->worker(); });

}

}

ThreadPool::~ThreadPool() {

{

std::unique\_lock<std::mutex> lock(tasksMutex);

stop = true;

}

condition.notify\_all();

for (auto& worker : workers) {

worker.join();

}

}

void ThreadPool::enqueueTask(std::function<void()> task) {

{

std::unique\_lock<std::mutex> lock(tasksMutex);

tasks.push(std::move(task));

}

condition.notify\_one();

}

void ThreadPool::worker() {

while (true) {

std::function<void()> task;

{

std::unique\_lock<std::mutex> lock(tasksMutex);

condition.wait(lock, [this] { return stop || !tasks.empty(); });

if (stop && tasks.empty()) return;

task = std::move(tasks.front());

tasks.pop();

}

task();

}

}

**2. Intrathread Communication**

* **Atomic Variables and Flags**: Use std::atomic<bool> to signal between threads. This ensures that state changes are visible across threads without the need for locking.
* **Condition Variables**: Use std::condition\_variable to notify threads about state changes.

**Example with Atomic Flag:**

cpp

#include <atomic>

#include <thread>

#include <iostream>

#include <chrono>

std::atomic<bool> stopFlag(false);

void calculations() {

while (!stopFlag.load()) {

// Perform calculations

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

std::cout << "Calculating..." << std::endl;

}

std::cout << "Calculations stopped." << std::endl;

}

void takeback() {

// Simulate user clicking 'Take back'

std::this\_thread::sleep\_for(std::chrono::seconds(2));

stopFlag.store(true);

std::cout << "Take back clicked!" << std::endl;

}

int main() {

std::thread calcThread(calculations);

std::thread takebackThread(takeback);

calcThread.join();

takebackThread.join();

return 0;

}

**3. Multithreading within Multithreads (Nested Threads)**

* **Thread Pools for Algorithms**: Using a thread pool for computationally intensive algorithms (like search algorithms) can significantly improve performance, especially on multi-core processors.
* **Granularity of Tasks**: Ensure that tasks are sufficiently granular to benefit from parallelism without causing too much overhead from context switching.

**Example with Nested Thread Pool:**

cpp

void searchTask(int id) {

std::cout << "Search task " << id << " running." << std::endl;

std::this\_thread::sleep\_for(std::chrono::milliseconds(500));

}

void complexCalculations(ThreadPool& pool) {

for (int i = 0; i < 5; ++i) {

pool.enqueueTask([i] { searchTask(i); });

}

}

int main() {

ThreadPool pool(std::thread::hardware\_concurrency());

std::thread calcThread(complexCalculations, std::ref(pool));

calcThread.join();

return 0;

}

By combining these strategies, you can create a robust and efficient multi-threaded Renju game. If you have further questions or need more specific implementations, feel free to ask! 😊📚💡