//实现一个简单线程池

#include <iostream>

#include <vector>

#include <thread>

#include<queue>

#include<mutex>

#include<future>

namespace **std**

{

//线程池最大容量,应尽量设小一点

#define THREADPOOL\_MAX\_NUM 16

//#define THREADPOOL\_AUTO\_GROW

//线程池,可以提交变参函数或拉姆达表达式的匿名函数执行,可以获取执行返回值

//不直接支持类成员函数, 支持类静态成员函数或全局函数,Opteron()函数等

class **threadpool**

{

public:

using Task = function<void()>; //定义类型，类似typedef

vector<thread> \_pool; //线程池

queue<Task> \_tasks; //任务队列

mutex \_lock; //同步

condition\_variable \_task\_cv; //条件阻塞， 需要配合 unique\_lock 使用，可以随时 unlock() 和 lock()

atomic<bool> \_run{ true }; //线程池是否执行，atomic为原子类型，不需要再加mutex

atomic<int> \_idlThrNum{ 0 }; //空闲线程数量

public:

inline **threadpool**(unsigned short size = 4)

{

unsigned short threadSize =

std::min<unsigned short>(size, std::thread::hardware\_concurrency());

addThread(threadSize);

}

inline ~**threadpool**()

{

\_run = false;

\_task\_cv.notify\_all(); // 唤醒所有线程执行

for (thread& thread : \_pool) {

//thread.detach(); // 让线程“自生自灭”，即分离主线程和子线程

if (thread.joinable()) //判断thread是否可以join或者detach

thread.join(); // 等待任务结束， 前提：线程一定会执行完

}

}

public:

// 提交一个任务

// 调用.get()获取返回值会等待任务执行完,获取返回值

// 有两种方法可以实现调用类成员，

// 一种是使用 bind： .commit(std::bind(&Dog::sayHello, &dog));

// 一种是用 mem\_fn： .commit(std::mem\_fn(&Dog::sayHello), this)

template<class F, class... Args>

auto **commit**(F&& f, Args&&... args)/\* ->future<decltype(f(args...))>\*/

{

if (!\_run) // stoped

throw runtime\_error("commit on ThreadPool is stopped.");

using RetType = decltype(f(args...)); // typename std::result\_of<F(Args...)>::type, 函数 f 的返回值类型

auto task = make\_shared<packaged\_task<RetType()>>(

bind(forward<F>(f), forward<Args>(args)...)

); // 把函数入口及参数,打包(绑定)

std::future<RetType> future = task->get\_future();

{ // 添加任务到队列

lock\_guard<mutex> lock{ *\_lock* };//对当前块的语句加锁 lock\_guard 是 mutex 的 stack 封装类，构造的时候 lock()，析构的时候 unlock()

\_tasks.push([task]() { // push(Task{...}) 放到队列后面 emplace

(\*task)();

});

}

#ifdef THREADPOOL\_AUTO\_GROW

if (\_idlThrNum < 1 && \_pool.size() < THREADPOOL\_MAX\_NUM)

addThread(1);

#endif // !THREADPOOL\_AUTO\_GROW

\_task\_cv.notify\_one(); // 唤醒一个线程执行

return future;

}

//空闲线程数量

int **idlCount**() { return \_idlThrNum; }

//线程数量

int **thrCount**() { return \_pool.size(); }

#ifndef THREADPOOL\_AUTO\_GROW

private:

#endif // !THREADPOOL\_AUTO\_GROW

//添加指定数量的线程

void **addThread**(unsigned short size)

{

//for (; \_pool.size() < THREADPOOL\_MAX\_NUM && size > 0; --size)

for (; size > 0; --size)

{ //增加线程数量,但不超过 预定义数量 THREADPOOL\_MAX\_NUM

\_pool.emplace\_back([this] { //工作线程函数

while (\_run)

{

Task task; // 获取一个待执行的 task

{

unique\_lock<mutex> lock{ *\_lock* }; // unique\_lock 相比 lock\_guard 的好处是：可以随时 unlock() 和 lock()

//task = move(\_tasks.front()); // 按先进先出从队列取一个 task

\_task\_cv.wait(*lock*, [this] {

return !\_run || !\_tasks.empty();

}); // wait 直到有 task

if (!\_run && \_tasks.empty())

return;

task = \_tasks.front();

\_tasks.pop();

}

\_idlThrNum--;

task();//执行任务

\_idlThrNum++;

}

});

\_idlThrNum++;

}

}

};

}

std::threadpool executor{80}; //使用

可以直接提交线程任务

如： ff[plevel] = executor.commit(std::bind(&TraceBlockData::toBlockThreadFunction, //任务

this, //其他数据

boxClassifyImg,

std::move(classImg[plevel]),

(plevel+1),

boxMaskHsvImg,

oriImg,

box,

boxIndex,

&boxTBlock,

p));