继续探索 C++ 中的多线程

迈克尔·阿代索

2019 年 11 月 16 日 警察

继续探索 C++ 中的多线程

系列文章

- 第 1 部分：探索 C++ 中的多线程
- **第 2 部分：探索 C++ 中的多线程续。**
- 第 3 部分：探索 C++ 中的多线程：加载纹理
- 第 4 部分：探索 C++ 中的多线程：并行化光线追踪

特定作业的特定工作线程

我的下一个测试用例是让不同的工作线程运行不同类型的任务。这个想法是有几个线程用于重要的工作，其他线程用于不太重要的工作。为简单起见，我已将任务拆分为不同的作业队列。

C++ 缩小▲ 复制代码

```
static std::mutex g_mutexLowJobQ;
static std::mutex g_mutexMediumJobQ;
static std::mutex g_mutexHighJobQ;

std::queue<CalcPiJob*> GetJobsOfType(int count, int iterations)
{
    std::queue<CalcPiJob*> jobQ;
    for (int i = 0; i < count; ++i)
    {
        jobQ.emplace(new CalcPiJob(iterations));
    }
    return jobQ;
}

void RunThreadedPriority()
{
    int nHighThreads = 3;
    int nMediumThreads = 2;
    int nLowThreads = 2;

    std::queue<CalcPiJob*> lowJobQ =
         GetJobsOfType(Settings::JobCountLow, Settings::IterationCountLow);
```

```
    std::queue<CalcPiJob*> mediumJobQ =
         GetJobsOfType(Settings::JobCountMedium, Settings::IterationCountMedium);
    std::queue<CalcPiJob*> highJobQ =
         GetJobsOfType(Settings::JobCountHigh, Settings::IterationCountHigh);

    std::vector<std::thread> threads;

    std::atomic<bool> hasHighJobsLeft = true;
    for (int i = 0; i < nHighThreads; ++i)
    {
        std::thread t([&]() {
            ExecuteJobsQ(hasHighJobsLeft, highJobQ, g_mutexHighJobQ);
        });
        threads.push_back(std::move(t));
    }

    std::atomic<bool> hasMediumJobsLeft = true;
    for (int i = 0; i < nMediumThreads; ++i)
    {
        std::thread t([&]() {
            ExecuteJobsQ(hasMediumJobsLeft, mediumJobQ, g_mutexMediumJobQ);
        });
        threads.push_back(std::move(t));
    }

    std::atomic<bool> hasLowJobsLeft = true;
    for (int i = 0; i < nLowThreads; ++i)
    {
        std::thread t([&]() {
            ExecuteJobsQ(hasLowJobsLeft, lowJobQ, g_mutexLowJobQ);
        });
        threads.push_back(std::move(t));
    }

    // main thread
    while (hasHighJobsLeft || hasMediumJobsLeft || hasLowJobsLeft)
    {
        if (hasHighJobsLeft)
        {
            ExecuteJobsQ(hasHighJobsLeft, highJobQ, g_mutexHighJobQ);
        }
        else
        {
            // wait for other threads to complete.
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
        }
    }

    const int threadCount = threads.size();
    for (int i = 0; i < threadCount; ++i)
    {
        threads[i].join();
    }
}
```

8 个线程的运行时间：6059 毫秒。（4 个高作业线程，2 个中线程和 2 个低线程。）

配置文件图像显示 4 个线程仅处理大型作业，2 个线程处理中等作业，另外 2 个线程处理较小作业。正如我们所看到的，这不会为我们赢得多少时间，因为当一些线程完成它们的工作时，它们处于空闲状态，对更大的图景没有贡献。

我们可以尝试通过实现某种工作窃取来解决这个问题。当一个线程没有更多的任务给他们时，他们可以从其他线程队列中窃取任务。

工作窃取的特定线程

下一个测试就是这样。每种线程类型都设置为在它们用完作业后从主线程中获取优先级较低的作业。希望我们能防止线程空闲。

C++ 缩小▲ 复制代码

```
void RunThreadedPriorityWorkStealing()
{
    int nHighThreads = 5;
    int nMediumThreads = 1;
    int nLowThreads = 1;

    std::queue<CalcPiJob*> lowJobQ =
         GetJobsOfType(Settings::JobCountLow, Settings::IterationCountLow);
    std::queue<CalcPiJob*> mediumJobQ =
         GetJobsOfType(Settings::JobCountMedium, Settings::IterationCountMedium);
    std::queue<CalcPiJob*> highJobQ =
         GetJobsOfType(Settings::JobCountHigh, Settings::IterationCountHigh);

    std::vector<std::thread> threads;

    std::atomic<bool> isHighPriorityThreadsActive = true;
    for (int i = 0; i < nHighThreads; ++i)
    {
        std::thread t([&]() {

            while (isHighPriorityThreadsActive)
            {
                CalcPiJob* currentJob = GetAndPopJob(highJobQ, g_mutexHighJobQ);

                // if no more High Jobs, take on Medium ones.
                if (!currentJob)
                {
                    currentJob = GetAndPopJob(mediumJobQ, g_mutexMediumJobQ);
                }

                // if no more Medium Jobs, take on Small ones.
                if (!currentJob)
                {
                    currentJob = GetAndPopJob(lowJobQ, g_mutexLowJobQ);
                }

                if (currentJob)
                {
                    currentJob->DoWork();
                    delete currentJob;
                }
                else
                {
                    isHighPriorityThreadsActive = false;
                }
            }
        });
        threads.push_back(std::move(t));
    }

    std::atomic<bool> isMediumThreadsActive = true;
    for (int i = 0; i < nMediumThreads; ++i)
    {
        std::thread t([&]() {
            while (isMediumThreadsActive)
            {
                CalcPiJob* currentJob = GetAndPopJob(mediumJobQ, g_mutexMediumJobQ);

                // if no more Medium Jobs, take on Small ones.
                if (!currentJob)
                {
```

```
                currentJob = GetAndPopJob(lowJobQ, g_mutexLowJobQ);
            }

            if (currentJob)
            {
                currentJob->DoWork();
                delete currentJob;
            }
            else
            {
                isMediumThreadsActive = false;
            }
        }
    });
    threads.push_back(std::move(t));
}

std::atomic<bool> isLowThreadsActive = true;
for (int i = 0; i < nLowThreads; ++i)
{
    std::thread t([&]() {
        while (isLowThreadsActive)
        {
            CalcPiJob* currentJob = GetAndPopJob(lowJobQ, g_mutexLowJobQ);

            if (currentJob)
            {
                currentJob->DoWork();
                delete currentJob;
            }
            else
            {
                isLowThreadsActive = false;
            }
        }
        });
    threads.push_back(std::move(t));
}

// main thread
while (isLowThreadsActive || isMediumThreadsActive || isHighPriorityThreadsActive)
{
    if (isHighPriorityThreadsActive)
    {
        CalcPiJob* currentJob = GetAndPopJob(highJobQ, g_mutexHighJobQ);

        // if no more High Jobs, take on Medium ones.
        if (!currentJob)
        {
            currentJob = GetAndPopJob(mediumJobQ, g_mutexMediumJobQ);
        }

        // if no more Medium Jobs, take on Small ones.
        if (!currentJob)
        {
            currentJob = GetAndPopJob(lowJobQ, g_mutexLowJobQ);
        }

        if (currentJob)
        {
            currentJob->DoWork();
            delete currentJob;
        }
        else
        {
            isHighPriorityThreadsActive = false;
        }
    }
    else
```

```
        {
            // wait for other threads to complete.
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
        }
    }

    const int threadCount = threads.size();
    for (int i = 0; i < threadCount; ++i)
    {
        threads[i].join();
    }
}
```

8 个线程的运行时间：2625 毫秒。

现在我们可以看到，一旦较高的工作线程耗尽，高优先级的工作线程就开始承担中等规模的工作，然后是小型工作。

同步线程

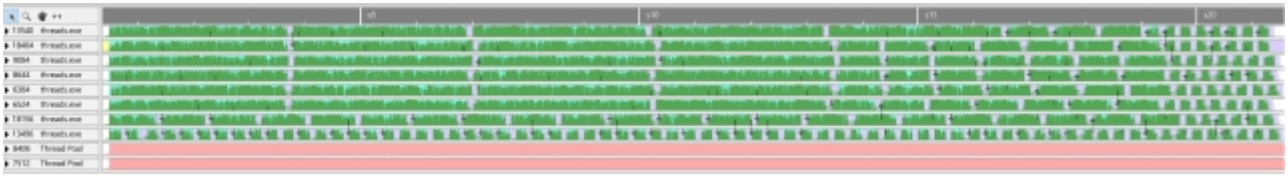
现在假设我正在处理数据并且我需要在多个线程之间同步启动作业，或者我可能正在构建一个游戏引擎并且我的主更新循环需要与其他线程中的物理循环同时启动. 无论哪种情况，我认为查找同步机制也值得一试。

C++ 缩小▲ 复制代码

```
std::mutex g_syncMutex;
std::condition_variable g_conditionVariable;

void RunSynchronizedThreads()
{
    int nThreads = std::thread::hardware_concurrency() - 1;
    std::vector<std::thread> threads;

    std::queue<CalcPiJob*> jobQ = GetJobsQ();

    std::atomic<bool> signal = false;
    std::atomic<bool> threadsActive = true;
    for (int i = 0; i < nThreads; ++i)
    {
        std::thread t([&]() {
            while (threadsActive)
            {
                // Tell main thread, worker is available for work
                {
                    std::unique_lock<std::mutex> lk(g_syncMutex);
                    g_conditionVariable.wait(lk, [&] { return signal == true; });
                }

                CalcPiJob* currentJob = GetAndPopJob(jobQ, g_mutexJobQ);

                if (currentJob)
                {
                    currentJob->DoWork();
                    delete currentJob;
                }
                else
                {
                    threadsActive = false;
                }
            }
        });
```

```
        threads.push_back(std::move(t));
    }

    // main thread
    std::atomic<bool> mainThreadActive = true;
    while (mainThreadActive && threadsActive)
    {
        // send signal to worker threads, they can start work.
        {
            std::lock_guard<std::mutex> lk(g_syncMutex);
            signal = true;
        }
        g_conditionVariable.notify_all();

        // send signal to worker threads, so they have to wait for their next update.
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
        {
            std::lock_guard<std::mutex> lk(g_syncMutex);
            signal = false;
        }
        g_conditionVariable.notify_all();

        // main thread work.
        CalcPiJob* currentJob = GetAndPopJob(jobQ, g_mutexJobQ);

        if (currentJob)
        {
            currentJob->DoWork();
            delete currentJob;
        }
        else
        {
            mainThreadActive = false;
        }
    }

    for (int i = 0; i < nThreads; ++i)
    {
        threads[i].join();
    }
}
```

运行时间：2674 毫秒



我已经设置了这个，因此工作线程仅以与主线程相同的频率启动。这里的目标是使用条件变量来同步线程，并希望用分析器确认它，我们可以在上图中看到。

测试运行	时间（毫秒）	改进
一根线	10396	1.99 倍
螺纹	2625	7.88 倍
线程优先	6059	3.4 倍
与工作窃取有关	2625	7.8 倍
同步线程	2674	7.7 倍

从 GitHub 下载代码

继续阅读

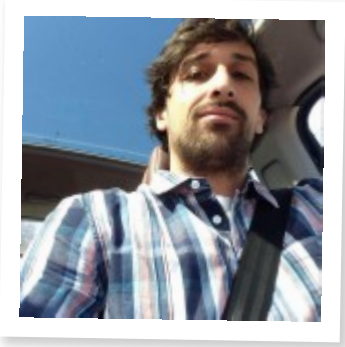
- 第 3 部分：探索 C++ 中的多线程：加载纹理
- 第 4 部分：探索 C++ 中的多线程：并行化光线追踪

本文最初发表于http://mikeadev.net/2019/10/exploring-multi-threading-in-c-part-2

执照

分享

关于作者



迈克尔·阿代索
软件开发人员
英国
手表该会员
http://mikeadev.net/about-me/

评论和讨论

添加评论或问题 电子邮件提醒 Search Comments

第一 页上一页 下一页

继续?
MarkTX 19-Nov-19 5:59

刷新 1

一般 新闻 建议 问题 错误 答案 笑话 赞美 咆哮 管理员

使用Ctrl+Left/Right 切换消息，Ctrl+Up/Down 切换主题，Ctrl+Shift+Left/Right 切换页面。