# Job System based on fiber（v0.1）

### 1.JobSystem概述

JobSystem是一种利用多线程技术加快执行job的系统。在现有计算机配置下，CPU往往拥有多个核心，可以同时执行大量的线程。为了利用这种物理特性，程序将大量重复的任务转换为一系列Job，然后系统再利用多线程技术去执行Job。

JobSystem最简化的一种模型就是生产者-消费者模型，最简单的实现就是实现一个工作线程池，一个线程安全的任务队列，每个线程会尝试从队列中获取Job/Task，然后程序则将各个任务以Job/Task的形式添加到队列中。类似的实现如下：

Void WorkerThread

{

Job job = null

While(GetJobFromQueue(job ))

{

If (job == null)

Wait(100)

else

Job()

}

}

Class JobSystem

{

Void AddJob(Job newJob)

{

mJobQueue.Enqueue(newJob)

}

ThreadSafeQueue mJobQueue;

}

基于上述实现的JobSystem就已经能够为系统的性能带来很大的提升，但是在现在程序规模越来越庞大的同时（例如游戏引擎），基于上述实现过于简单的JobSystem不但性能存在问题，应用的场景也存在局限。

其中一个主要的问题在于，目前各个Job之间的优先级关系是平行的，但是实际上部分任务应该拥有更高的优先级，这就意味着我们需要实现多级队列，同时当需要执行优先级更高的当前Job在执行完前无法yield。

另一些需要考虑的问题在于如何尽量让每个worker thread都执行尽可能多的job，或者说是各个线程之间的平衡，尽量较少单独线程等待的时间。通过Job Stealing是一种可以减少这种问题的方法，但也会因此引申出其他问题，类似于多线程之间一些的冲突问题。

总之是说明如上述简单实现的jobSystem是远远不够，一个可用的JobSytem是需要考虑许多方面，下面会基于顽皮狗GDC的分享，实现一个简单的可用的Fiber based job system

### Fiber

Fiber意为纤程，是一种类似如协程（coroutine）轻量级线程，是基于Win32系统下独有的类型。纤程的主要的两个特点如下：

1. 纤程拥有独立的栈空间和寄存器空间
2. 纤程是由用户态实现调度
3. 一个线程上只有一个纤程正在执行

通过上述两个特点，如果了解coroutine的话，会发现是非常类似的。Fiber因为完全由用户控制，所以Fiber的切换也是由用户代码控制的。代用Fiber的switchToFiber会从当前纤程切换到另一个纤程，而当前纤程的状态都会由纤程独立的栈空间和寄存器保存。综上所述，Fiber和thread的区别，就是在于Fiber是由用户态控制，完全可以视为coroutine

一个典型的Fiber封装如下：

Clsss Fiber

{

      Fiber(EntryPointFunc entryPointFunc, void\* userData, I32 stackSize = DEFAULT\_STACK\_SIZE,);

        Fiber(ThisThread, std::string debugName = "");

        ~Fiber();

        void SwitchTo();

        void\* GetUserData();

        bool IsValid()const;

        static Fiber\* GetCurrentFiber();

}

上述Fiber主要提供了两种构造方法，一种是直接创建一个Fiber同时包含了一个Fiber执行的函数，另一个方法则是把当前的线程转换为Fiber。

    Fiber::Fiber(EntryPointFunc entryPointFunc, void\* userData, I32 stackSize)

    {

        mImpl->fiber\_ = ::CreateFiber(stackSize, FiberEntryPoint, mImpl);

}

    Fiber::Fiber(ThisThread, std::string debugName)

    {

        mImpl->fiber\_ = ::ConvertThreadToFiber(mImpl);

    }

Fiber提供了SwithTo方法主要是对系统API::switchToFiber的封装调用。同时当Fiber切换时，会记录下当前的线程运行的Fiber（使用Fiber专用的存储结构FLS保存），staic方法GetCurrentFiber则返回上述保存的Fiber。

以上就是Fiber的概念和封装实现，基本理解为coroutine即可。

### 基于Fiber的JobSystem

接下来我们就可以基于Fiber为Job unit来创建JobSystem。

因为Fiber用于独立的栈空间和寄存器，且在用户态切换，意味着可以yield当前Fiber而切换到其他Fiber，也就能够实现暂停当前job，而先去执行优先级更高的job，当优先级更高的job完成后，则再切回到之前的job执行，这也就弥补了上文我们所说的普通thread job system的缺点。

根据描述，我们定义JobSystem的数据结构如下：

std::vector<WorkerThread\*> mWorkerThreads; // 工作相乘

MPMCBoundedQueue<JobFiber\*> mFreeFibers; // 当前可用的Fiber

std::array<MPMCBoundedQueue<JobInfo>, (I32)Priority::MAX> mPendingJobs; // 基于优先级的工作队列

std::array<MPMCBoundedQueue<JobFiber\*>, (I32)Priority::MAX> mWaitingFibers;// 基于优先级的等待Fiber

JobSystem的大致思路如下：

1.启动一定数量的workerThread，（数量一般为CPU线程数-1）workerThread启动的同时会将自身转换为Fiber并设置为线程的current fiber

2.创建一定数量的JobFiber，存放在Fiber中

3.mWaitingFiber初始时为空，mPending则根据优先级放入pending jobs

4.workerThread每次会依次从高优先级到底优先级：

4.1 从pendingJobs中拿去job,如果job存在，则尝试从FreeFiber中拿去可用的Fiber，如果无可用Fiber，则会yield直到等待可用的FreeFiber，拿到FreeFiber后，则将当前job设置给Fiber，同时代用Fiber::switchTo，从workerThread(也是workerFiber）切换到jobFiber，jobFiber会尝试去执行job，当执行完后则将Fiber放入到FreeFiber中。

注意这里会出现我们前文说过的情况，可能因为存在更高优先级的job，当前Job过程中执行了Yield操作，yield操作会取得当前执行的fiber（Fiber::GetCurrentFiber），并将fiber标记为等待，然后切回到workFiber（workerThread），此时workThread则会将这个jobFiber放入对应优先级的waitingFibers队列中

4.2 如果当前优先级未存在job，则会尝试从waitingFiber中拿去waitingFiber，然后swichTo这个waitingFiber，继续执行之前Yield的job。

5.使用JobSystem时，主线程可能需要等待一系列的job都完成，这里使用Counter来实现这个功能，Counter就是一个当前任务数量的计数，创建job时会将Counter指针保存到job中，每当job完成时，则会将计数减一，而主线程则在计数不为0时一直等待。

基于上述实现思路，实现的代码如下：

### 参考

<https://zhuanlan.zhihu.com/p/36309461> [GDC15]Parallelizing the Naughty Dog Engine using Fibers