线程与线程池

作者：WTF

原文链接：https://ld246.com/article/1613662194105

来源网站：链滴

<h2 id="1-CPU">1、CPU</h2>
CPU 并不知道线程进程之类的概念
CPU 只知道两件事：

从内存中取出指令
执行指令，然后回到 1


Q1：CPU 从哪里取出指令？
PC 寄存器（Program Counter），也就是程序计数器，可以把寄存器理解为存取速度更快的存。
Q2：PC 寄存器中存放的是什么？
存放的是指令在内存中的地址，这里的指令是 CPU 将要执行的下一条指令。

Q3：谁来设置 PC 寄存器中的指令地址？
原来 PC 寄存器中的地址是默认自动加 1 的，因为大部分情况下 CPU 都是一条接一条的顺序执行当遇到 if、else 时，这种顺序执行就被打破了，CPU 在执行这类指令时会根据计算结果来动态改变 PC 寄存器的值，这样 CPU 就可以正确的跳转到需要执行的指令了。
Q4：PC 寄存器中的初始值是怎么被设置的？
CPU 执行的指令来自内存，内存中的指令是从磁盘中保存的可执行程序加载过来的，磁盘中的可执程序是编译器生成的，编译器从我们定义的函数中生成的机器指令。

函数被编译后才会形成 CPU 执行的指令，那么我们该如何让 CPU 执行一个函数呢？，显然我们需找到函数被编译后的第一条指令就可以了，第一条指令就是函数入口。
<h2 id="2-从CPU到操作系统">2、从 CPU 到操作系统</h2>
我们想让 CPU 执行某个函数，只需要把函数对应的第一条级其执行装入 PC 寄存器就可以了，样即使没有操作系统我们也可以让 CPU 执行程序，虽然可行但是非常繁琐。
我们需要：

在内存中找到一块大小合适的区域装入程序
找到函数入口，设置好 PC 寄存器让 CPU 开始执行程序

机器指令需要加载到内存中执行，因此需要记录下内存的起始地址和长度，同时需要找到函数的口地址并写到 PC 寄存器中。想一想需要一个数据结构来记录下这些信息。
数据结构大致如下：
<pre><code class="language-cpp highlight-chroma">
</code></pre>

ight-p">};
</code></pre>
这个结构体用来记录程序在被加载到内存中的运行状态，程序从磁盘加载到内存跑起来叫进程 Process。
CPU 执行的第一个函数叫 main 函数。
完成上述两个步骤的程序被称作操作系统 Operating System。
<h2 id="3-从单核到多核-如何充分利用多核-">3、从单核到多核，如何充分利用多核？</h2>
Q1：假设我们想写一个程序并且要充分利用多核该怎么办？
 多开几个进程？
 有道理，但是存在问题：

进程需要占用内存空间，如果多个进程基于同一个可执行程序，那么这些进程在内存区域中的内几乎完全相同，这显然会造成内存的浪费。
计算机处理的任务可能是比较复杂的，涉及到了进程间通信，由于各个进程处于不同的内存地址间，进程间通信天然需要借助操作系统，既增大了编程难度也增加了系统开销。

<h2 id="4-从进程到线程">4、从进程到线程</h2>
所谓进程无非就是内存中的一段区域，这段区域保存了 CPU 执行的机器指令以及函数运行时的栈信息，要想让进程运行，就把 main 函数的第一条机器指令地址写入 PC 寄存器，这样进程就运行来了。
进程的缺点在于只有一个入口函数，也就是 main 函数，因此进程中的机器指令只能被一个 CPU 执行。
Q1：有没有办法让多个 CPU 来执行同一个进程中的机器指令？
 既然我们可以把 main 函数的第一条指令地址写入 PC 寄存器，那么其他函数和 main 函数有什么别呢？没　什么区别，main 函数的特殊之处无非就是 CPU 的第一个函数，我们可以把 PC 寄存器指 main 函数，就可以把 PC 寄存器指向任何一个函数。
当我们把 PC 寄存器指向非 main 函数时，线程就诞生了。
一个进程内可以有多个入口函数，也就是说属于同一个进程中的机器指令可以被多个 CPU 同时行。
注意：这是一个和进程不同的概念，创建进程时我们需要在内存中找到一合是的区域以装入进程，然后把 CPU 的 PC 寄存器指向 main 函数，也就是说进程中只有一个执行。
但是现在不一样了，多个 CPU 可以在同一个屋檐下（进程占用的内存区域）同时执行属于该进的多个入口函数，也就是说现在一个进程内可以有多个执行流了。执行流也就是线程。
操作系统为每个进程维护了一堆信息，用来记录进程所属的内存空间等，这对信息记为数据集 A
同样的，操作系统也需要为线程维护一堆信息，用来记录线程的入口函数或者堆栈信息等，这对据记为数据集 B。
显然数据集 B 要比数据集 A 的量要少，同时不像进程，创建一个线程时无需去内存中找一段内空间，因为线程时运行在所处进程的地址空间的，这块地址空间在程序启动时已经创建完毕，同时线是程序在运行期间创建的（进程启动后），因此当线程开始运行的时候这块地址空间就已经存在了，以直接使用。这就是创建线程比创建进程快的原因（还有其他原因）。
有了线程的概念后，我们只需在进程开启后创建多个线程就可以让所有 CPU 忙起来，这就是所高性能、高并发的根本所在。
注意：由于各个线程共享进程的内存地址空间，因此线程之间的通信无需借助操作系统，这给程员带来极大方便的同时也带来了无尽的麻烦，多线程遇到的多数问题都出自于线程间的通信太方便了出错的根源在于 CPU 执行指令时根本没有线程的概念，多线程编程面临的互斥与同步问题需要程序自己解决，...
提醒：单核的情况下也可以创建出多个线程，线程是操作系统层面的实现和有多少个核心没有关系，CPU 在执行机器指令时也意识不到执行的机器指令属于哪个线程。一个 CU，操作系统也可以通过线程调度让各个线程同时向前推进，方法就是将 CPU 的时间片在各个线程间回分配，看起来同时运行，但实际上任意时刻只有一个线程在运行。
<h2 id="5-线程与内存">5、线程与内存</h2>
把 CPU 的 PC 寄存器指向线程的入口函数，这样线程就可以运行起来了，这就是为什么我们创

线程时必须指定入口函数的原因。
<pre><code class="language-cpp highlight-chroma"><span cl
ss="highlight-cl">// 设置线程入口函数DoSomething
<span cla
s="highlight-c1">thread <span class="highligh
-o">= CreateThread
DoSomething);</sp
n>

<span class="high
ight-c1">// 让线程运行起来
<span cla
s="highlight-c1">thread<span class="highlight
p">.Run();
</code></pre>
Q1：线程和内存有什么关联？
 函数在执行时产生的数据包括：函数参数，局部变量，返回地址等信息。这些信息是保存在栈中的
线程这个概念还没有出现时，进程中只有一个执行流，因此只有一个栈，这个栈的栈底就是进程的入
函数 main 函数。
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2021/02/image-accc3210.png?imageView2/2/interlace/1/format/jpg"></
>
有了线程以后，一个进程中就存在多个执行入口，即同时存在多个执行流，只有一个执行流的进
需要一个栈来保存运行时信息，那么很显然有多个执行流就需要多个栈来保存各个执行流的信息，也
是说操作系统需要为每个线程在进程的地址空间中分配一个栈，即每个线程都有独属于自己的栈。</
>
创建线程需要消耗进程内存空间。
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2021/02/image-f21e4842.png?imageView2/2/interlace/1/format/jpg"></
>
<h2 id="6-线程的使用">6、线程的使用</h2>
从生命周期的角度将，线程要处理的任务有两类：长任务和短任务。

长任务 long-lived tasks
 任务存活的时间很长，如 word，我们在 word 中编辑的文字需要保存在磁盘上，往磁盘上写数据
是一个任务，这时比较好的办法就是专门创建一个写磁盘的线程，该写线程的生命周期和 word 进程
一样的，dakaiword 就创建写线程，关闭 word 时才销毁。

 这种场景非常适合创建专用的线程来处理某些特定任务。
<ol start="2">
短任务 short-lived tasks
  任务处理的时间很短，比如一次网络请求，一次数据库查询，这种任务可以在短时间内快速处理完
。因此段任务多见于各种 server，如 web server，database server，file server 等。

 特点：
  任务处理所需时间短，
  任务数量巨大
Q1：如何处理这种类型的任务？
 当 server 接收到一个请求后就创建一个线程来处理任务，处理完成后销毁该线程。
 这种方法被称为 thread-per-request，一个请求就创建一个线程。
 如果是长任务，这种方法很好，
 但是对于大量的短任务，这种方法虽然实现简单但是有缺点：
  1. 线程是操作系统中的概念，因此创建线程天然需要借助操作系统完成，操作系统创建销毁
程是需要消耗时间的。

□□□□ 2. 每个线程需要自己独立的栈，因此创建大量线程会消耗过多的内存等系统资源。
这就好比你是一个工厂老板（想想都很开心有没有），手里有很多订单，每来一批订单就要招一工人，生产的产品非常简单，工人们很快就能处理完，处理完这批订单后就把这些千辛万苦招过来的人辞退掉，当有新的订单时你再千辛万苦的招一遍工人，干活儿 5 分钟招人 10 小时，如果你不是励要让企业倒闭的话大概是不会这么做的。
因此更好的策略就是招一批人后就地养着，有订单时处理订单，没订单时闲着。
这就是线程池的由来。
<h2 id="7-线程池">7、线程池</h2>
创建一批线程，之后就不再释放，有任务就交给这些线程处理，因此无需频繁的创建销毁线程，程个数通常是固定的，也不会消耗过多的内存，这里的思想就是复用、可控。
Q1：怎么给线程池提交任务？这些任务又是怎么给到线程池中的线程？
□□ 数据结构中的队列天然适合这种场景，提交任务的就是生产者，消费线程的就是消费者。
□□ * [ ] 待看：生产者消费者
一般来说提交给线程池的任务包含两部分：
□□ 1. 需要被处理的数据
□□ 2. 处理数据的函数
伪代码：struct 可以理解为 class
<pre><code class="language-cpp highlight-chroma"><span cl
ss="highlight-cl">struct tas
 {
  <span class="h
ghlight-kt">void* da
a; // 任务所携带
数据
<span cla
s="highlight-c1">
  handler <span class="highli
ht-n">handle; //
理数据的方法
<span cla
s="highlight-c1">
}
</code></pre>
线程池中的线程会阻塞在队列上，当生产者向队列中写入数据后，线程池中的某个线程会被唤醒将该线程从队列中取出上述结构体，以结构体中的数据为参数并调用处理函数。
<pre><code class="language-cpp highlight-chroma"><span cl
ss="highlight-cl">while(</sp
n>true) <span clas
="highlight-p">{
  <span class="hi
hlight-k">struct task <span class="highlight-o
>= GetFromQueue()
 // 从队列中取出数据
<span cla
s="highlight-c1">
  task<span class="highlight-o
>->handle(</spa
>task-><span cla
s="highlight-n">data); <span class="highligh
-c1">// 处理数据
<span cla
s="highlight-c1">
}
</code></pre>
<h2 id="8-线程池中线程的数量">8、线程池中线程的数量</h2>
要知道线程池中的线程过少就不能充分利用 CPU，线程创建的过多反而会造成系统性能下降，存占用过多，线程切换造成的消耗等。因此线程的数量不能太多也不能太少。
线程池处理的任务：

□□ 从生命周期看：长任务、短任务
□□ 从处理任务所需的资源角度看：CPU 密集型和 I/O 密集型
□□ □□ 1. CPU 密集型
□□ □□ □□ 处理任务不需要依赖外部 I/O，如科学计算，矩阵运算等。这种情况下只要线程的数量和核基本相同就可以完全充分利用 CPU。
□□ □□ 2. I/O 密集型
□□ □□ □□ 理论上 2N 个线程，具体情况具体分析。
线程池仅仅时多线程的一种使用形式，因此多线程面临的问题线程池同样不能避免，像死锁问题 race condition？问题等，参考操作系统相关资料，打好基础！！！
<h2 id="9-线程池使用的最佳实践">9、线程池使用的最佳实践</h2>
使用前需考虑：

充分理解你的任务，是长任务还是短任务，CPU 密集型还是 I/O 密集型，如果都有，可能更好办法是把两类任务放到不同的线程池中，这样也许可以更好的确定线程数量。
如果线程池中的任务有 I/O 操作，务必设置超时，否则该线程可能会一直阻塞。
线程池中的任务最好不要同步等待其他任务的结果。

【整理】【原文：http://www.52im.net/thread-3272-1-1.html 】
<pre><code class="highlight-chroma">
</code></pre>
<pre><code class="highlight-chroma">
</code></pre>
<pre><code class="highlight-chroma">
</code></pre>