多线程编程

概述

每个线程是 CPU 使用的一个基本单元； 它包括线程 ID 、 程序计数器 、 寄存器组和堆栈。 它与同一进程的其他线程共享代码段、 数据段和其他操作系统资源，

大多数的操作系统内核现在都是多线程的。 多个线程在内核中运行， 每个线程执行一个特定任务， 如管理设备 、 管理内存或处理中断。

四大类的优点：响应性，资源共享，经济，可伸缩性

多 核 编 程

无论多个计算核是在多个 CPU 芯片上还是在单个 CPU 芯片上， 我们称之为多核或多处理器系统。 多线程编程提供机制， 以便更有效地使用这些多个计算核和改进的并发性。

并行性 和并发性 的区别。

并行系统可以同时执行多个任务。 相比之下， 并发系统支持多个任务， 允许所有任务都能取得进展。 因此，没有并行，并发也是可能的

编程挑战

多核系统编程有五个方面的挑战：1、识别任务，2、平衡，3、数据分割，4、数据依赖，5、测试与调试

并行类型

数据并行 注重将数据分布于多个计算核上， 并在每个核上执行相同操作。

任务并行涉及将任务（线程） 而不是数据分配到多个计算核。 每个线程都执行一个独特的操作。

多线程模型

用户层的用户线程 或内核层的内核线程 用户线程位于内核之上,它的管理无需内核支持； 而内核线程由操作系统来直接支持与管理。

多对一模型

多个用户级线程到一个内核线程

线程管理是由用户空间的线程库来完成的， 因此效率更高

不过， 如果一个线程执行阻塞系统调用， 那么整个进程将会阻塞。 再者， 因为任一时间只有一个线程可以访问内核， 所以多个线程不能并行运行在多处理核系统上

一对一模型

每个用户线程到一个内核线程

提供了比多对一模型更好的并发功能，它也允许多个线程并行运行在多处理器系统上

唯一缺点是，创建一个用户线程就要创建一个相应的内核线程，由于创建内核线程的开销会影响应用程序的性能，所以这种模型的大多数实现限制了系统支持的线程数量

多对多模型

多路复用多个用户级线程到同样数量或更少数量的内核线程

开发人员可以创建任意多的用户线 程， 并且相应内核线程能在多处理器系统上并发执行。 而且， 当一个线程执行阻塞系统调用时， 内核可以调度另一个线程来执行。

多对多模型的一种变种仍然多路复用多个用户级线程到同样数量或更少数量的内核线程 但也允许绑定某个用户线程到一个内核线程，称为双层模型

线 程 库

线程库为程序员提供创建和管理线程的API

实现线程库的两种方法

在用户空间中提供一个没有内核支持的库，这种库的所有代码和数据结构都位于用户空间。 这意味着， 调用库内的一个函数只是导致了用户空间内的一个本地函数的调用， 而不是系统调用。

第二种方法是， 实现由操作系统直接支持的内核级的一个库。 对于这种情况， 库内的代码和数据结构位于内核空间。 调用库中的一个 API 函数通常会导致对内核的系统调用

三 种 主 要 线 程 库 是： POSIX Pthreads、 Windows、 Java。

隐式多线程

将多线程的创建与管理交给编译器和运行时库来完成。 这种策略称为隐式线程

线程池

在进程开始时创建一定数量的线程， 并加到池中以等待工作。 当服务器收到请求时， 它会唤醒池内的一个线程 （ 如果有可用线程）， 并将需要服务的请求传递给它。 一旦线程完成了服务， 它会返回到池中再等待工作。 如果池内没有可用线程， 那么服务器会等待， 直到有空线程为止。

优点:
• 用现有线程服务请求比等待创建一个线程更快。
• 线程池限制了任何时候可用线程的数量。 这对那些不能支持大量并发线程的系统非
常重要。
• 将要执行任务从创建任务的机制中分离出来， 允许我们采用不同策略运行任务。
例如， 任务可以被安排在某一个时间延迟后执行， 或定期执行。

OpenMP

OpenMP 为一组编译指令和 API, 用于编写 C、 C++、 Fortran 等语言的程序，它支持共享内存环境下的并行编程。OpenMP 识别并行区域, 即可并行运行的代码块。

大中央调度

Apple Mac OS X 和 iOS 操作系统的一种技术， 为 C 语言、 API 和运行时库的一组扩展， 它允许应用程序开发人员将某些代码区段并行运行。

GCD 为 C 和 C++ 语言增加了块 (block) 的扩展。

通过将这些块放置在调度队列 （ dispatchqueue) 上， GCD 调度块以便执行。 当 GCD 从队列上移除一块后， 就将该块分配给线程池内的可用线程。 GCD 识别两种类型的调度队列： 串行 (serial) 和并发 (concurrent)0

多线程问题

系统调用 forkO 和 execO

有的 UNIX 系统有两种形式的 forkO, 一种复制所有线程， 另一种仅仅复制调用了系统调用 forkO 的线程。

系统调用 eXeC()的工作方式与第 3 章所述方式通常相同。 也就是说， 如果一个线程调用 execO 系统调用， execO 参数指定的程序将会取代整个进程， 包括所有线程。

信号处理

用于通知进程某个特定事件已经发生。

遵循相同的模式:
• 信号是由特定事件的发生而产生的。
• 信号被传递给某个进程。
• 信号一旦收到就应处理。

同步信号发送到由于执行操作导致这个信号的同一进程

例子包括非法访问内存或被 0 所除。

当一个信号是由运行程序以外的事件产生的， 该进程就异步接收这一信号。

例子包括使用特殊键 （ 比如 <c0ntr0l>＜C>) 来终止进程， 或者定时器到期等。

信号处理程序

缺省信号处理程序

用户定义信号处理程序

信号应被传递到哪里

通常具有如下选择:
• 传递信号到信号所适用的线程。
• 传递信号到进程内的每个线程。
• 传递信号到进程内的某些线程
• 规定一个特定线程以接收进程的所有信号。

线程撤销

是在线程完成之前终止线程。

需要撤销的线程称为目标线程。 目标线程的撤销可以有两种情况:

异步撤销： 一个线程立即终止目标线程。
• 延迟撤销： 目标线程不断检查它是否应终止， 这允许目标线程有机会有序终止自己。

异步撤销

通常， 操作系统收回撤销线程的系统资源， 但是并不收回所有资源，因此， 异步撤销线程可能不会释放必要的系统资源

延迟撤销

对于延迟撤销， 一个线程指示目标线程会被撤销； 不过， 仅当目标线程检查到一个标志以确定它是否应该撤销时， 撤销才会发生。 线程可以执行这个检查： 它是否位于安全的撤销点

缺省撤销类型为延迟撤销。

这样， 只有当线程到达撤销点 时， 才会发生撤销。 建立撤销点的一种技术是， 调用函数ptHreacLtestcancelO。 如果有一个撤销请求处于等待， 那么就会调用称为清理处理程序 的函数。 在线程终止前，这个函数允许释放它可能获得的任何资源

线程本地存储

某些情况下， 每个线程可能需要它自己的某些数据 ， 我们称这种数据为线程本地存储

TLS 与局部变量容易混淆。 然而， 局部变量只在单个函数调用时才可见； 而 TLS 数据在多个函数调用时都可见。 在某些方面， TLS 类似于静态 数据。 不同的是，TLS数据是每个线程独特的。

调度程序激活

涉及内核与线程库间的通信

许多系统在实现多对多或双层模型时， 在用户和内核线程之间增加一个中间数据结构，这个数据结构通常称为轻量级进程

用户线程库与内核之间的一种通信方案称为调度器激活

它工作如下： 内核提供一组虚拟处理器 （ LWP) 给应用程序， 而应用程序可以调度用户线程到任何一个可用虚拟处理器。

此外， 内核应将有关特定事件通知应用程序。 这个步骤称为回调 , 它由线程库通过回调处理程序 来处理。

当一个应用程序的线程要阻塞时， 一个触发回调的事件会发生。 在这种情况下，内核向应用程序发出一个回调， 通知它有一个线程将会阻塞并且标识特定线程。 然后， 内核分配一个新的虚拟处理器给应用程序。 应用程序在这个新的虚拟处理器上运行回调处理程序， 它保存阻塞线程的状态， 并释放阻塞线程运行的虚拟处理器。