线程

相关概念

定义

- 定义：线程是独立的指令流，可以被内核调度运行
- 进程包含的状态和资源
  - 代码、堆、数据、文件句柄（包括套接字）、进程间通信（IPC）
  - 进程ID、进程组ID、用户ID
  - 栈、寄存器、程序计数器（PC）
- 线程与进程关系
  - 线程存在于进程内部，并共享进程的资源
  - 每个线程都有自己的核心资源（线程独有资源）
    - 栈
    - 寄存器
    - 程序计数器
    - 线程特定数据
  - 访问共享资源需要同步
  - 线程由内核独立调度
  - 每个线程都有自己独立的控制流
  - 每个线程都可以处于任何调度状态

线程的优势

- 响应性：多线程交互应用程序允许程序即使在部分被阻塞或执行长时间操作时也能继续运行
- 资源共享：资源共享可以实现高校通信和高度协作。线程默认共享进程的资源和内存。
- 经济性：线程比进程更轻量级，创建和上下文切换的开销更小

多线程服务器架构

架构示意图

server

* 工作方式解释：server的主进程（一般是监听进程）接受客户端的请求，然后主进程创建一个新的线程来处理请求。之后主监听进程继续监听其他的客户端请求

注：并发与并行

定义

- 并发计算是一种计算形式，其中程序被设计为相互交互的计算进程集合，这些进程可以并行执行。并发程序（进程或线程）可以在单个处理器上通过时间片轮转的方式交错执行各自的执行步骤，也可以通过将每个计算进程分配给一组处理器来并行执行。程序作为独立执行进程的组合，这些进程相互通信。
- 行计算是一种计算形式，其中许多计算同时进行，基于大问题通常可以分解为较小问题的原理，然后"并行"解决这些小问题。
  编程作为（可能相关的）计算的同时执行

对比

维度	并发 (Concurrency)	并行 (Parallelism)
核心概念	同一时间段内交替执行	同一时刻真正同时执行
硬件要求	单核CPU即可	必须多核CPU或多台机器
设计思想	如何组织和管理任务	如何真正同时计算
主要目的	提高响应性、资源利用率	提高计算速度

关键理解

- 并发关乎结构，并行关乎执行
- 并发提供了一种构建解决方案的方式来解决一个可能（但不一定）可并行化的问题

线程的实现

基本信息

- 线程可以在用户级别或由内核提供
- 用户线程在内核之上支持，无需内核支持即可管理
  - 三个线程库：POSIX Pthreads、Win32线程和Java线程
- 内核线程由内核直接支持和管理
  - 所有现代操作系统都支持内核线程

线程实现的方式

内核级线程

定义

- 为了使并发更经济，进程的执行方面被分离为线程。因此，操作系统现在管理线程和进程。所有线程操作都在内核中实现，操作系统调度系统中的所有线程。由操作系统管理的线程成为内核级线程。
- 在这种方法中，内核知道并管理线程。这种情况下不需要运行系统。内核不是在每个进程中保持线程表，而是拥有一个线程表来跟踪系统中的所有线程。此外，内核还维护传统的进程表来跟踪进程。操作系统内核提供系统调用来创建和管理线程。

优势

- 因为内核完全了解所有线程，调度器可能决定给拥有大量线程的进程分配更多时间，而不是给拥有少量线程的进程。
- 内核级线程对于频繁阻塞的应用程序特别有效

劣势

- 内核级线程速度慢而且效率低
- 存在显著的开销和内核复杂性的增加

用户级线程

定义

- 用户级显示完全由运行时系统（用户级库）管理内核对用户级线程一无所知，将它们作为单线程进程管理。用户级线程小而快，每个线程由PC、寄存器、栈和小型线程控制块表示。创建新线程、线程间切换和线程同步都通过过程调用完成，即无内核参与。用户及线程比内核级线程块100倍。

优势

- 这种技术最明显的优势是用户级线程包可以在不支持的线程的操作系统上实现
- 用户级线程不需要修改操作系统
- 简单表示：每个线程简单地由PC、寄存器、栈和小型控制块表示，都存储在用户进程地址空间中
- 简单管理：这意味着创建线程和线程间同步都可以在没有内核干预的情况下完成
- 快速高效：线程切换比过程调用贵不了多少

劣势

- 用户级线程不是完美解决方案，它们是一种权衡。由于用户级线程对操作系统不可见，它们与操作系统集成不好。结果是，操作系统可能做出糟糕决策，如调度有空闲线程的进程、阻塞启动I/O的线

程所在的进程（即使该进程有其他可运行线程）、取消调度持有锁的线程所在的进程。解决者需要内核和用户级线程管理器之间的通信。
- 线程和操作系统内核之间缺乏协调。因此，整个进程获得一个时间片，无论进程有一个线程还是1000个线程。每个线程都要主动放弃控制权给其他线程。
- 用户级线程需要非阻塞调用，即多线程内核。，否则，整个进程将在内核中阻塞，即使进程中还有可运行的线程。

多线程模型

含义：

- 用户线程和内核线程之间必须存在关系
- 内核线程是系统中真正的线程，所以为了使用户线程取得进展，用户程序必须让其调度器获取一个用户线程，然后再内核线程上运行它
- 核心：只有内核线程真正在CPU上执行

多线程模型详解

多对一模型(Many-to-one)

- 多个用户级线程映射到单个内核线程
- 线路管理由用户空间的线程库完成
- 如果一个线程进程阻塞系统调用，整个进程将被阻塞
- 将阻塞系统调用转换为非阻塞
- 多个线程无法再多处理器上并行运行

一对一模型(One-to-one)

- 每个用户级线程映射到一个内核线程
- 允许其他线程在一个线程阻塞时运行
- 多个线程可以在多处理器上并行运行
- 总计金额导致开销
- 大多数实现次模型的操作系统限制线程数量

多对多模型

- 许多用户级线程映射到许多内核线程
- 解决了1:1和m:1模型的缺点
- 开发人员可以创建必要数量的用户线程

- 相应的内核线程可以在多处理器上并行运行

两级模型

- 类似对于多对多模型，除了它允许用户线程绑定到内核线程

fork和exec的语义问题

fork

- 当对单线程进程fork时，直接复制整个单线程进程
- 当读多线程进程fork时，可以理解为只复制调用线程或复制所有线程，在UNIX中有两个版本的fork，每种语义一个

exec

- exec通常替换整个进程

fork与exec的综合使用

- 如果在fork后很快调用exec，使用“fork调用线程版本”，不需要复制所有线程

信号处理

概念

- 信号在UNIX系统中用于通知进程发生了特定事件，遵循相同的模式
- 信号由特定事件的发生而产生
- 信号倍传递给进程
- 一旦传递，信号必须被处理

原则

- 信号由两种信号处理程序之一处理：默认的或用户定义的
- 每个信号都有默认处理程序，内核在处理信号时运行
- 用户定义的信号处理程序可以覆盖默认的
- 对于单线程，信号传递给进程

多线程环境下的信号处理

- 信号可以是同步的（异常）或异步的（I/O）

- 同步信号传递给引起信号的同一线程
- 一部信号可以传递给
  - 信号适用的线程
  - 进程中的每个线程
  - 进程中的某些线程（信号掩码）
  - 接收进程所有信号的特定线程

线程取消

概念

- 线程取消：在目标线程完成之前终止它

取消的实现方法

- 异步取消：立即终止目标线程
- 延迟取消：允许目标线程定期检查是否应该被取消

取消的实现细节

- 调用线程取消请求取消，但是实际取消取决于线程状态
- 如果线程禁用了取消，取消保持挂起状态直到线程启用它
- 默认类型时延迟取消
- 取消只在线程到达取消点时发生
  - 即pthread_testcancel()
- 然后调用清理处理程序
- 在Linux系统上，线程取消通过信号处理

线程特定数据(Thread Specifif Data)

线程特定数据

- 线程本地存储(TLS)允许每个线程拥有自己的数据副本
- 当你无法控制线程创建过程时很有用（即使用线程池时）
- 与局部变量不同
- 局部变量只在单个函数调用期间可见
- TLS在函数调用间可见
- 类似于静态数据

- TLS对每个线程都是唯一的

轻量级进程与调度器激活

轻量级进程概念

- 在计算机操作系统中，轻量级进程（LWP）是实现多任务的一种方式。在传统意义上，如Unix System V和Solaris中使用的术语，LWP在用户空间中运行在单个内核线程之上，并与同一进程内的其他LWP共享地址空间和系统资源。多个用户级线程由线程库管理，可以放置在一个或多个LWP之上——允许在用户级进行多任务处理，这可以带来一些性能优势
- 在一些操作系统中，内核线程和用户线程之间没有单独的LWP层。这意味着用户线程直接在内核线程之上实现。在这些情况下，术语"轻量级进程"通常指内核线程，而术语"线程"可以指用户线程。在Linux上，用户线程通过允许某些进程共享资源来实现，这有时导致这些进程被称为"轻量级进程"

调度方式

- 轻量级进程(LWP)是多对多和两级模型中用户线程和内核线程之间的中间数据集二狗
- 对用户线程库来说，它看起来像虚拟处理器来调度用户线程
- 每个LWP都连接到一个内核线程
- 内核线程阻塞 -> LWP阻塞 -> 用户线程阻塞
- 内核调度内核线程，线程库调度用户线程
- 线程库可能做出次优的调度决策
- 解决方案：让内核通知线程库的重要的调度事件
- 调度器激活通过上调通知线程库

Windos XP线程

概念

- Windos XP实现一对一映射线程模型
- 每个线程包含
  - 线程ID
  - 处理器状态的寄存器集
  - 堵路的用户栈和内核栈
  - 私有数据存储区域
- 线程的主要数据结构包括
  - ETHREAD：执行线程块（

- KTHREAD：内核线程块
- TEB：线程环境块

Linux线程

- linux有fork和clone两个系统调用
- clone接收一组标志位，决定父进程和子进程之间的共享程度
- FS/VM/SIGHAND/FILES -> 相当于线程创建
- 没有设置标志 -> 没有共享 -> 相当于fork
- Linux不区分进程和线程，使用术语“任务”而不是线程

线程库

概念

- 线程库为程序员提供了创建和管理线程的API接口

两种主要实现方式

- 用户空间实现
  - 完全在用户空间中实现，无需内核支持
  - 特点：快速、轻量级，但无法利用多核
  - 示例：早期的Green Threads
- 内核级实现
  - 由操作系统支持的内核级库
  - 特点：可以真正并行，但开销较大
  - 示例：现代操作系统的标准实现

线程特性

- 线程拥有自己的身份标识，并且可以独立运行
- 线程共享进程内的地址空间，享受避免任何进程间通信(IPC)通道（共享内存、管道等）进行通信的好处
- 进程中的线程可以直接相互通信
- 例如独立的线程可以访问/更新全局变量
- 这种模型消除了内核本来需要承担的潜在IPC开销。由于线程在同一地址空间中，线程上下文切换是廉价且快速的

Pthread调度

- API允许在线程创建时指定PCS或SCS调度范围
- pthread_attr_set/getscope是相关的API
- PTHREAD_SCOPE_PROCESS：使用PCS调度来调度线程
- LWP的数量由线程库维护
- PTHREAD_SCOPE_SYSTEM：使用SCS调度来调度线程
- 可用的调度范围可能受到操作系统的限制
- 例如：Linux和Mac OS X只允许PTHREAD_SCOPE_SYSTEM

多处理器调度

多处理器架构类型

- 多处理器可能是以下任一架构
  - 多核CPU
  - 多线程核心
  - NUMA系统
  - 异构多处理
- 多处理器调度基础
  - 当有多个CPU可用时，CPU调度变得更加复杂
  - 假设处理器在功能上时相同的（同构的）
  - 多处理器调度的方法
  - 非堆成多处理：
    - 只有一个处理器做调度决策、I/O处理和其他活动
    - 其他处理器充当虚拟处理单元
  - 对称多处理(SMP)：每个处理器都是自调度的
    - 调度数据结构是共享的，需要同步
    - 被通用操作系统使用
- SMP架构细节
  - 对称多处理(SMP)是每个处理器都自调度的架构
  - 所有线程可能在一个公共就绪队列中(a)
  - 或者每个处理器可能有自己的私有线程队列(b)
- 多核调度
  - 单芯片中的多个CPU核心
  - 最近的趋势是在同一物理芯片上防止多个处理器核心，更快且功耗更低
- 芯片多线程(CMT)

    - 多线程核心：芯片多线程
    - Intel使用超线程术语（或同时多线程-SMT）：在同一核心上同时运行两个（或更多）硬件线程：内存停顿
    - 利用内存停顿在内存检索时在另一个线程上取得进展
    - 每个核心有>1个硬件线程。如果一个线程有内存停顿，切换到另一个线程！
- CMT的两级调度
    - 两级调度：
        - 操作系统决定在逻辑CPU上运行哪个软件线程
        - 每个核心如何决定在物理核心上运行哪个硬件线程。两个硬件线程不能并行运行，因为我们只有一个CPU核心
        - 如果操作系统知道CPU资源的底层共享情况，可以做出更好的决策
- 负载均衡
    - 如果是SMP，需要保持所有CPU的负载以提高效率
    - 负载均衡试图保持工作负载均匀分布
    - 推送迁移 - 周期性任务检查每个处理器的负载，如果发现则将任务从过载的CPU推送到其他CPU
    - 拉取迁移 - 空闲处理器从繁忙处理器拉取等待任务
- 处理器亲和性
    - 当线程在一个处理器上运行时，该处理器的缓存内容存储该线程的内存访问
    - 我们称这为线程对处理器有亲和性（即"处理器亲和性"）
    - 负载均衡可能影响处理器亲和性，因为线程可能从一个处理器移动到另一个处理器以平衡负载，但该线程失去了在其移出的处理器缓存中的内容
    - 软亲和性 - 操作系统试图保持线程在同一处理器上运行，但不保证
    - 硬亲和性 - 允许进程指定一组它可以运行的处理器
- NUMA和CPU调度
    - 如果操作系统是NUMA感知的，它将分配靠进展线程运行CPU的内存
- 实时CPU调度
    - 可能出现明显的挑战
    - 软实时系统-关键实时任务有最高优先级，但不保证任务何时被调度
    - 硬实时系统-任务必须在其截止时间前得到服务

Linux 2.6.23+ 完全公平调度器(CFS)详解

- Linux调度器版本2.6.23+
    - 完全公平调度器(CFS)
    - 调度类
    - 每个调度类都有特定的优先级
    - 调度器选择最高优先级调度类中的最高优先级任务

    - 不是基于固定时间分配的量子，而是基于CPU时间比例(nice值)
    - 较少的nice值将获得更高比例的CPU时间
    - 包含2个调度类，其他可以添加
        - 默认调度类
        - 实时调度类
- CFS量子计算细节
    - 量子基于nice值计算，从-20到+19
    - 较低的值是更高的优先级
    - 计算目标延迟 - 任务应该至少运行一次的时间间隔
    - 如果活跃任务数量增加，目标延迟可以增加
    - CFS调度器在变量vruntime中维护每个任务的虚拟运行时间
    - 与基于任务优先级的衰减因子相关联 - 较低优先级有较高的衰减率
    - 正常默认优先级产生虚拟运行时间 = 实际运行时间
    - 要决定下一个运行的任务，调度器选择虚拟运行时间最低的任务

Linux调度系统补充详解

- Linux实时调度
    - 根据POSIX.1b标准的实时调度
    - 实时任务具有静态优先级
    - 实时任务加上普通任务映射到全局优先级方案
    - Nice值-20映射到全局优先级100
    - Nice值+19映射到优先级139
- Linux负载均衡与NUMA
    - Linux支持负载均衡，但也是NUMA感知的
    - 调度域是一组可以相互平衡的CPU核心集合
    - 域按它们共享的内容（即缓存内存）组织。目标是防止线程在域之间迁移

Windows调度系统详解

- Windows调度基础
    - Windows使用基于优先级的抢占式调度
    - 最高优先级的线程下一个运行
    - 调度器就是分发器(Dispatcher)
    - 线程运行直到 (1)阻塞，(2)用完时间片，(3)被更高优先级线程抢占
    - 实时线程可以抢占非实时线程
    - 32级优先级方案：可变类是1-15，实时类是16-31
    - 优先级0是内存管理线程

    - 每个优先级一个队列
    - 如果没有可运行线程，运行空闲线程
- Windows优先级类
    - 不同的优先级类
    - 一个类内的相对优先级