深入浅出UNIX系统内核（一）

一、进程与内核

1、内核是什么？

内核是一个特殊的程序，可以直接运行在设备上。他实现了进程模型和其他系统服务。内核驻留在磁盘中的一个文件中。当系统启动时，使用称为引导的特殊过程从磁盘上加载内核。接着，内核初始化系统，为进程运行设置环境。然后，创建几个初始进程，这些进程随后依次创建其他进程，一旦加载完成，内核会一直驻留在内存中，直到系统关闭。他管理着所有的进程，为进程提供各种服务。

下图展示了MAC OS & iOS的XNU内核

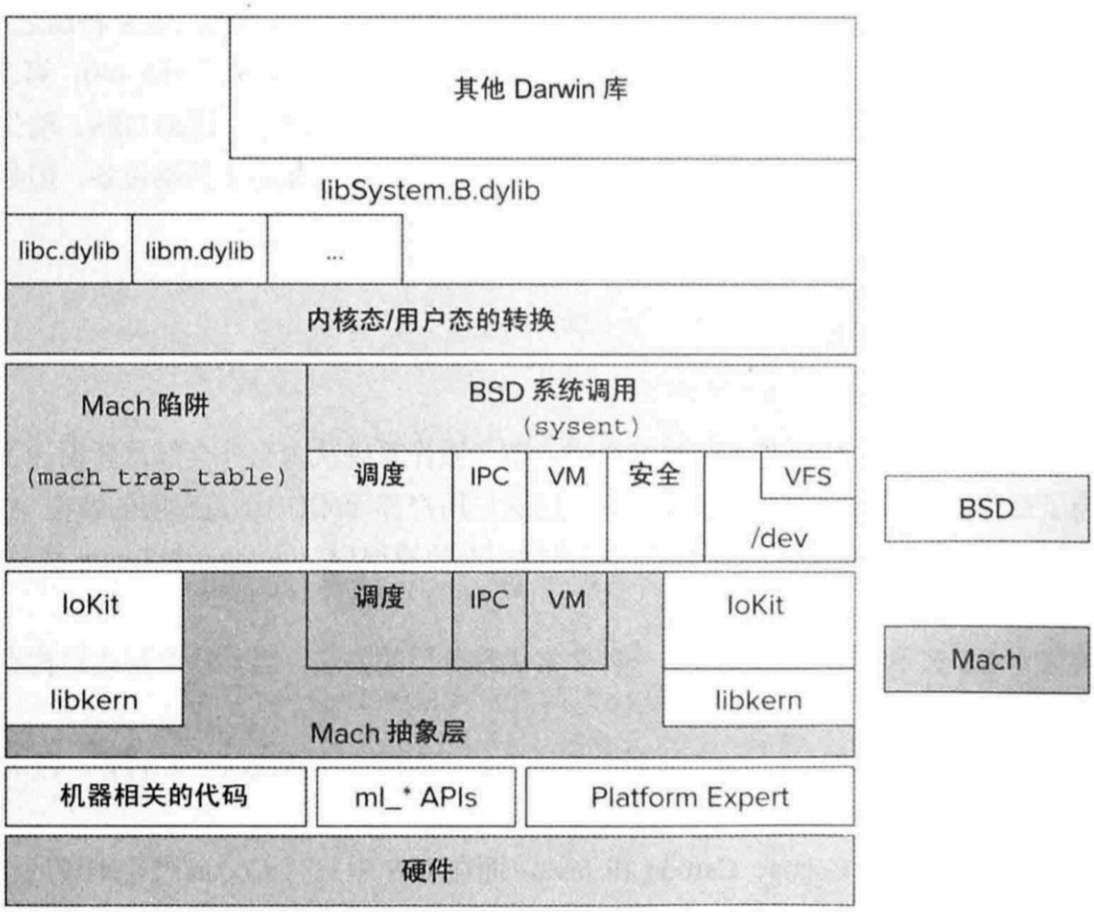

图 2-2　Darwin 架构

2、虚拟内存

在虚拟内存系统中，程序使用的地址不会直接引用物理内存中的位置，每个进程都有自己的虚拟内存地址空间，通过使用一组地址转换映射表，将虚拟内存地址的引用转换为物理内存中的位置。计算机的内存管理单元（MMU）通常会有一组寄存器来标识当前正在运行进程的转换映射表。在当前进程将CPU让给另外一个进程时（即发生了上下文切换），内核会将这些寄存器和指针加载到新进程的转换映射表中。MMU寄存器是受特权保护的，只能在内核态下访问。这就确保一个进程只能引用它自己空间中的地址，而不能访问或者修改其他进程的地址空间。
每个进程的虚拟地址空间有一个固定的部分映射为内核的代码和数据结构内容。该部分称为系统空间或内核空间，只能在内核态下访问。在系统中，因为只有一个内核实例在运行，所有进程都映射到了单个内核地址空间上。

3、内核态与用户态

内核是一个受信任的系统组件，控制着最为关键的功能。内核的功能和应用程序的功能之间需要有一种严格的分离，否则，应用程序的不稳定可能会使整个系统崩溃。

内核态和用户态可以理解为程序的运行权限，系统调用和中断处理都会将程序由用户态转换为内核态。

inter架构 — ring

所有的操作系统内核都会再启动的时候将CPU切换为保护模式。

保护模式强制的使用了4个
“ring”。这些”ring”指的就是权限级别，分别从0到3编号。这些“ring”以同心圆的方式组织，最内层的ring为ring0，最外层的ring为ring3.ring0的权限最高，通常被称为超级用户模式。只有最受信任的代码才能运行在处理器的ring0上，这些代码几乎无所不能。随着ring级别的递增，安全限制越多，权限也越低，因此最不受信任的代码运行在ring3上，这些代码受到的限制也是最多的。

ring0对应的是内核态，ring3对应的是用户态。ring1和ring2预留给操作系统服务使用。

ring的区分是通过CS寄存器中的两个位以及EFLAGS寄存器中对应的两个位实现的，CS寄存器中的两个位表示“用户的权限级别”，EFLAGS寄存器中的两个位表示当前的权限级别，这些标志位都属于线程状态的一部分。

ARM架构 — CPSR

ARM处理器使用了一个特殊的寄存器—当前程序状态寄存器（CPSR），来定义处理器所在的模式。ARM处理器有不少于7种不同的操作模式

USR（用户模式—不允许特权操作）

SVC（管理器模式—默认的内核模式）

SYS（系统模式—同用户模式，但允许写入CPSR）

FIQ（快速中断请求）

IRQ（普通中断请求）

ABT（中止模式—错误的内存访问）

UND（未定义模式—非法/不支持的指令）

USR是唯一没有特权的模式。所有其他模式都是特权模式，内核通常运行在SVC模式。在任何特权模式中，都可以直接访问CPSR寄存器，因此只要修改CPSR中的模式位即可切换模式。在用户态，必须使用一种用户态/内核态转换机制。

4、进程

进程是一个实体，运行一个程序并为它提供一个可执行的环境。进程包含一个地址空间和一个控制点，进程是基本的调度实体，在CPU上一次只能运行一个进程。（UNIX进程模型）

控制点：进程的控制点通过使用一个叫程序计数器的硬件寄存器来跟踪指令序列。很多后期的UNIX版本在单个进程中支持多个控制点（称为线程），因此一个进程内可以支持多个指令序列

进程状态

fork系统调用创建一个新的进程，最开始是初始状态，完全创建完成后，fork将其转换到就绪状态，在这里进程必须等待被调度。最后内核选择了执行该进程，发起一个上下文切换，将进程的硬件上下文加载到系统寄存器中，并将控制权转给进程。（调度）

当进程被安排调度运行时，它首先运行在内核态（内核运行状态）中，在这里完成上下文切换。下一个转换取决于进程被切换出去之前所做的事情。如果进程是被新创建的或者正在执行用户的代码（并由于为了让更高优先级的进程先运行而被取消调度），会立刻回到用户态。如果他在执行某个系统调用而被阻塞，它会在内核态下恢复该系统调用的执行。

最后，进程调用exit系统调用或者由于信号（signal，内核所发出的通知）而终止。 — 进程处于僵死状态

父进程调用wait（或者其他wait变种）后，销毁进程，退出状态返回给进程。

下图为进程状态和状态转换：

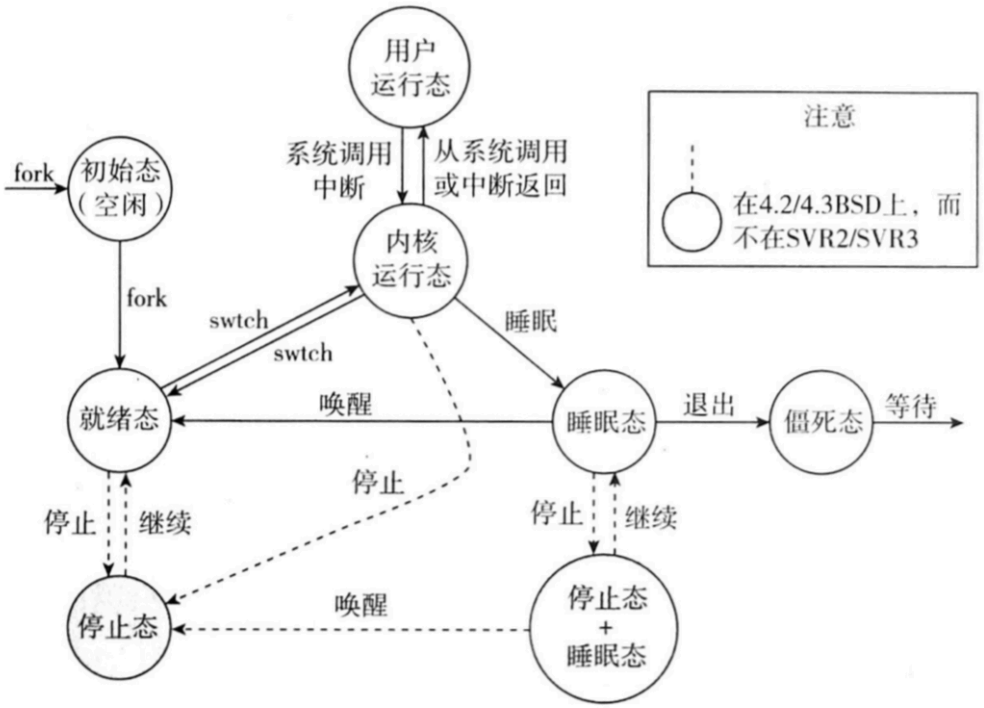

图 2-3　进程状态和状态转换

二、线程与轻量级进程

进程模型的局限性

1. 很应用程序希望并发的执行一些大型的独立任务，但又必须与其他进程共享同一个公共的地址空间和其他资源。
2. 传统的进程无法充分利用多处理器体系架构的优点，因为在一个时刻只能使用一个处理器。

很多应用程序必须执行大量独立且不需要串行化的任务。如果系统可以提供子任务并发执行的机制，这些应用程序将会运行的更好。

线程

线程表示进程中的一个控制点，并执行一系列指令。进程中的所有线程共享着相同的地址空间。

每个线程还有自己的私有对象，如程序计数器，栈和寄存器上下文。

几种重要的不同类型的线程

1、内核线程

内核线程不需要与用户进程关联，它由内核内部根据需要创建和销毁，负责执行一个特殊的功能。内核线程共享了内核的代码和全局数据，有自己自己的内核栈。内核线程可以被独立的调度，和使用标准的内核同步机制，如sleep()和wakeup()。

内核线程并不是一个新的概念。传统的UNIX内核中的pagedaemon这样的系统进程在功能上与内核线程是等价的。

2、轻量级进程(LWP)

轻量级进程，是由内核支持的用户线程，它是基于内核线程更高级别的抽象，因此一个系统，必须在可以支持LWP之前支持内核线程，每个进程可以有一个或多个LWP，每个LWP由一个单独的内核线程来支持。调度器通过调度内核线程来调度LWP，LWP共享进程的地址空间和其他资源。

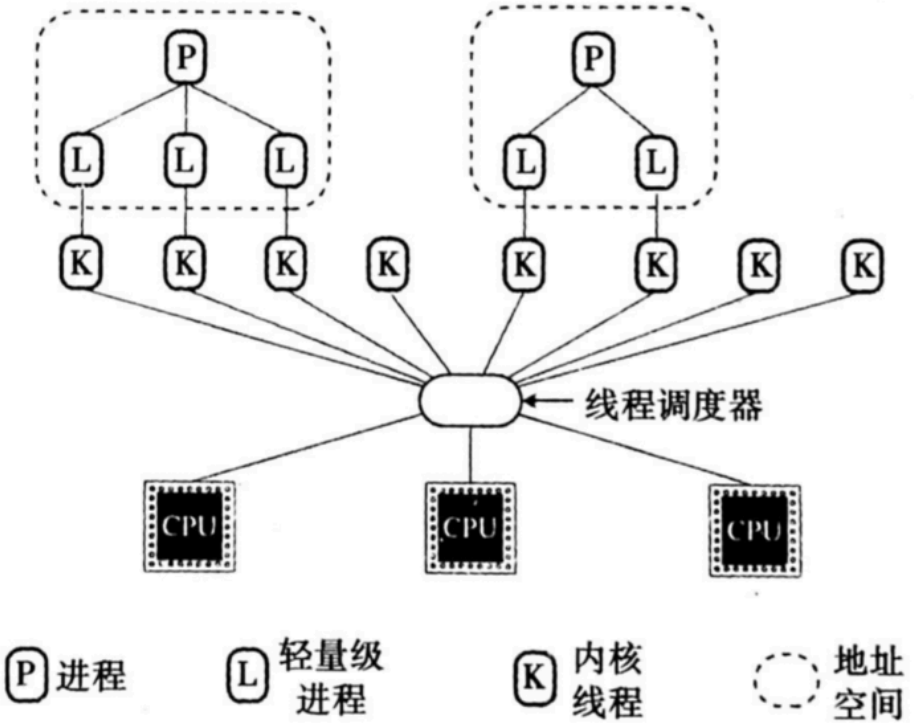

图 3-4　轻量级进程

局限性

1、大部分的LWP操作，如创建、析构和同步，都需要系统调用。系统调用相对而言是开销很大的操作，因为每个系统调用需要两个模式切换：一个切换发生在调用时从用户态到内核态，另一个切换是在完成之后返回用户态。LWP横跨了一个保护边界。内核必须从用户空间中复制系统调用参数到内核空间中，并进行验证，防止恶意的或者有bug的进程的访问。同样的，从系统调用返回时，用户必须将数据复制回用户空间。

2、当LWP频繁访问共享数据时，同步的开销会抵消掉任何性能好处。（大部分处理器系统提供了这样的锁机制）

每个LWP都要花费相当多的内核资源（内核栈），包括供内核栈使用的物理内存。因此，系统无法支持大量的LWP，

3、用户线程

线程抽象概念完全可以在用户级别上阐述，而不需要内核了解线程的任何东西。这是通过库来实现的，如Mach的C-thread和POSIX pthread。这些库提供了创建、同步、调度和管理线程的所有函数，且与内核没有特殊的关联。线程交互不设计内核，因而速度非常快。

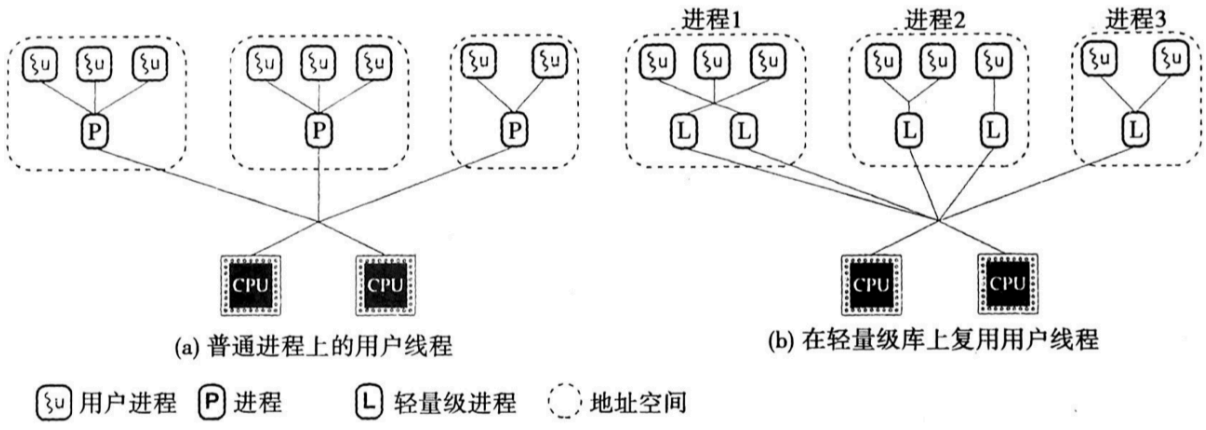

图 3-5　用户线程的实现

图a：普通进程上的用户线程。

图b：结合用户线程和轻量级进程，内核负责识别、调度和管理LWP。用户级别的库在LWP上层复用用户线程，并提供线程间调度、上下文切换和同步的工具，而不需要内核的参与。

库在用户线程间调度和切换上下文是通过保存当前线程的栈和寄存器内容，然后在加载新调度线程的相关内容来完成的。

内核仍然负责进程切换，因为只有内核具有修改内存管理寄存器内容的权限。内核调度线程下层的进程或LWP，这些进程或者LWP在使用库函数来调度它的线程。

库也提供了同步对象来保护共享的数据结构。（锁机制）

好处

性能，用户线程十分轻量，不会消耗内核资源。

局限性

1、内核与线程库之间信息完全分离，内核无法使用它的保护机制来保护线程，而且线程库必须提供同步工具。

2、由于这种分割调度模型，线程库负责调度用户线程，内核负责调度底层的进程或者LWP，而且彼此之间不知道对方在做什么。

例如：因为内核不清楚用户线程的相对优先级，他可以抢占一个运行着高优先级用户线程的LWP来调度一个运行着低优先级的用户线程的LWP。

3、LWP可能会在它的用户线程运行一个阻塞式的系统调用时阻塞在内核中，进程可能不会去运行LWP，即使是在有可运行的线程和可用的处理器资源时。

4、没有显示的内核支持，用户线程可以提升并发能力，但无法提升并行能力。

调度器激活（新的线程体系结构）

（由SGI的厂商的商业线程库使用）

它组合了两个模型的优点，其基本原则是让用户线程和内核之间紧密结合。内核负责处理器的分配，线程库负责调度。线程库把那些影响处理器分配的事件通知给内核。

如果线程阻塞在内核中，进程并不会失去处理器。内核会通知线程库，线程库随后立即调度另外一个线程在该处理器上运行。这个实现需要两个新的抽象：向上调用（upcall）和调度器激活（scheduler activation）

向上调用是指内核对线程库的调用。

调度器激活是可以用来运行用户线程的可执行上下文。它与LWP类似，也有自己的内核栈和用户栈。当内核执行一个向上调用时，内核传递了一个调度器激活给线程库，线程库用调度器激活来处理事件、运行新线程或调用另外一个系统调用。内核不会再某个处理器上对调度器激活进行时间分片。在任何时候，每个进程在分配给它的处理器上只会有一个调度器激活。

调度器激活框架的一个显著特性是它对阻塞操作的处理。当用户线程阻塞在内核中的时候，内核创建一个新的调度器激活和几个对线程库的向上调用。线程库保存了来自旧调度器激活中的线程状态，并通知内核旧的调度器激活可以被重用。线程库接着在新的调度器激活上调度另外一个用户线程。当阻塞完成时，内核创建另外一个向上调用把该事件通知给线程库。这个向上调用需要一个新的调度器激活，内核可能会分配一个新的处理器来运行这个激活，或者抢占该进程的某个当前激活。在后一种情况下，向上调用会通知线程两个事件：（1）原始的线程可以恢复运行，（2）在那个处理器上运行的线程已经被抢占。线程库将两个线程都放在就绪队列上，并决定哪个会得到调度。

调度器激活有很多优点。因为大多数操作不需要内核的参与，所以他们的运行速度非常的快。

Solaris 和 SVR4 上的多线程

同时支持内核线程、轻量级进程、用户线程

Mach的线程

Mach同时支持内核中的线程和用户级别库的线程。

Mach也支持处理器集合的概念，系统上可用的处理器可以被分为不重叠的处理器集合。每个任务和线程可以被分配到任意处理器集合。这允许多处理器系统上的一些CPU专用于一个或多个特定的任务，从而保证高优先级任务的资源需求。

Mach的抽象：任务和线程

任务

任务（task）是一个静态的对象，由一个地址空间和一组叫端口权利（port right）的系统资源集合组成。任务本身并不是一个可执行的实体，仅仅是可供一个或多个线程执行的环境。

线程

线程是基本的执行单元，运行在任务的上下文中。每个任务可以包含零个或多个线程，线程都共享着任务的资源。每个线程有一个内核栈，用

于系统调用处理，线程还有自己的计算状态（程序计数器、栈指针、通用寄存器等），而且是被处理器独立的调度。属于用户任务的线程等价于轻量级进程（每个用户线程会在底层关联一个内核线程uthread），纯内核线程属于内核任务。

Mach的C-threads

Mach提供了一个叫C-threads的库，提供了一个简易的接口来创建和管理线程。

C-threads库有三种实现：

1. 基于协同例程：在单线程的任务（UNIX进程）上复用用户线程。
2. 基于线程：每个C-thread使用一个不同的Mach线程。这些线程是可以被抢占调度的，可以在多处理器上并行的执行。这是默认的实现，用于C-thread程序的生产环境。
3. 基于任务：每个C-thread使用一个Mach任务（UNIX进程）。

Digital UNIX

基于Mach2.5内核

多线程进程同时得到了内核与POSIX兼容线程库的支持。UNIX进程是基于Mach的任务和线程之上抽象之上实现的。

Mach 3.0的continuation

原有Mach的局限性

原有Mach使用用户线程关联内核线程的方式，虽然内核线程比进程更轻量，但它仍然消耗大量的内核内存，这主要用在它的栈（内核栈）上。内核栈一般需要消耗4k字节的内存，在一个拥有大量线程的系统上，这种开销就会变得很大。

一种解决方式是在Mach线程或轻量级进程上复用用户线程。（避免为每个线程分配一个内核栈）

Mach线程或轻量级进程上复用用户线程的局限性

用户线程不能被独立的调度，因此不能提供同一级别上的并发。

因为内核线程没有跨越任务边界，每个任务必须包含至少一个内核线程，这会在有很多活动任务的系统上产生问题。

内核的编程模型：

进程模型（UNIX内核）

每个线程有一个内核栈，在线程由于系统调用或异常而陷入内核的时候会被使用到。当线程阻塞在内核中时，这个栈包含了它的执行状态（调用序列，自动变量），阻塞结束时，恢复执行。

优点：简单，内核线程阻塞时不用显示的保存任何状态。

缺点：消耗了大量的内存。

中断模型（操作系统QuickSilver和V）

内核将系统调用和异常当做中断，每个处理器有个内核栈用于所有的内核操作。因此，如果一个线程在内核中阻塞时，它必须显示的将它的状态保存到其他地方。在下次运行时，内核使用这个保存的信息来恢复线程的状态。

优点：只有一个单独的内核栈，可以节省很多内存。

缺点：线程必须为每个潜在的阻塞操作显示的保存它的状态信息。这使得难以使用该模型，因为必须保存的那些信息可能会跨越模块边界。因此，如果线程在一个深度嵌套的过程中阻塞时，他必须决定调用链上所有函数需要的状态信息。

Mach3.0 的 continuation 工具结合了这两个模型的优点，允许内核根据环境选择阻塞的方法。

直接好处：减少系统中内核栈的数目。大大减少了对内核内存的需求。

三、进程调度

四、进程间通讯（IPC）

参考书籍：

《深入理解UNIX系统内核》

《深入解析MAC OS & iOS 操作系统》