第 4 章：线程与并发性

第 4 章：线程

- 概述 多核编程 多线程模型
- 线程问题 操作系统示例
-
-
-

目标

- 识别线程的基本组件，并对比线程和进程。
- 描述设计多线程进程的主要好处和挑战。
- 描述 Windows 和 Linux 操作系统如何表示线程

动机

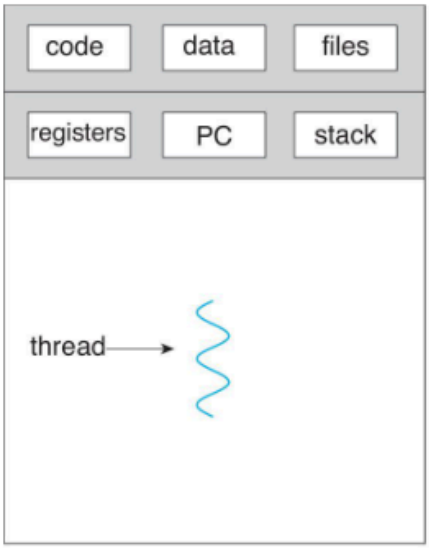
- 许多应用程序或程序都是多线程的
  - Web 浏览器可能有一个线程显示图像或文本，而另一个线程从网络检索数据
  - 文字处理器可能有一个用于显示图形的线程，另一个用于响应用户击键的线程，以及用于在后台执行拼写和语法检查的第三个线程
- 大多数操作系统内核都是多线程的
  - 例如，在 Linux 系统启动期间，会创建多个内核线程。每个线程执行特定的任务，例如管理设备、内存管理或中断处理
- 它可以利用多核系统上的处理能力
  - 并行编程广泛应用于数据挖掘、图形和人工智能等应用
- 流程创建既耗时又占用资源
  - 进程创建是重量级的，线程创建是轻量级的，属于同一进程的不同线程共享代码、数据等

多线程程序示例

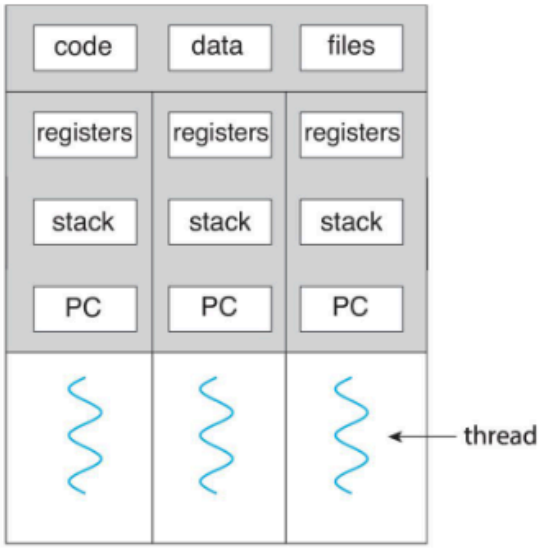
- 嵌入式系统
  - 电梯、飞机、医疗系统、手表 单程序、并发操作
  -
- 最现代的操作系统内核
  - 内部并发处理多个用户的并发请求，但内核内部不需要保护
  -
- 数据库服务器
  - 许多并发用户访问共享数据还必须完成后台实用程序处理
  -
- 网络服务器
  - 来自网络的并发请求 再次，单个程序，多个并发操作
  - 文件服务器、Web 服务器和机票预订系统
  -
- 并行编程
  - 将程序和数据拆分为多个线程以实现并行性

单线程和多线程进程

- 回想一下，线程是 CPU 使用的基本单位——独立调度和运行（称为执行实例），由线程 ID、程序计数器 (PC)、寄存器组和堆栈表示。它与同一进程的其他线程共享其代码、数据和其他操作系统资源，例如打开的文件和信号



single-threaded process



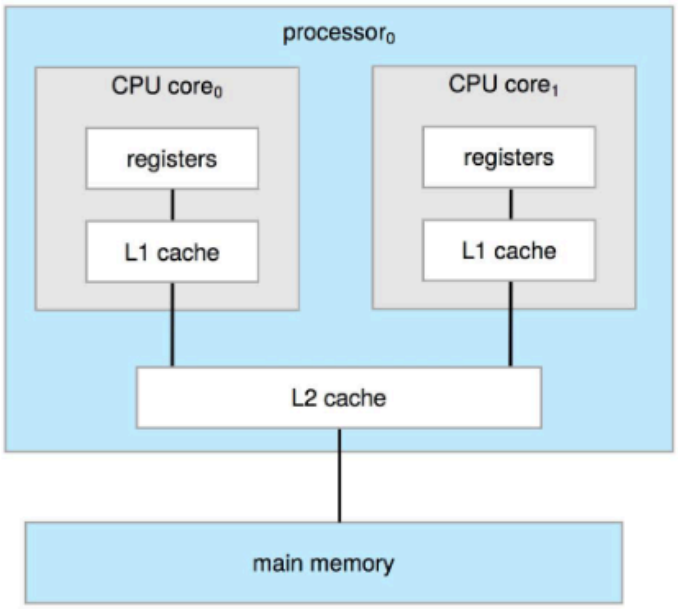
multithreaded process

好处

- 响应性 – 如果进程的一部分（例如进程的一个线程）被阻止，则可以允许继续执行，这对于具有用户界面的交互式应用程序尤其重要
- 资源共享 – 线程默认共享进程的资源，比共享内存或进程之间的消息传递更容易 – 因为它们本质上运行在进程的相同地址空间内 经济 – 线程创建比进程创建“便宜”得多（消耗更少的时间和内存），进程的线程之间的上下文切换通常比进程之间的上下文切换更快
- 可扩展性——进程可以利用多核架构

多核设计

- 多线程编程提供了一种更有效地使用多核的机制，并提高了多核系统的并发性。

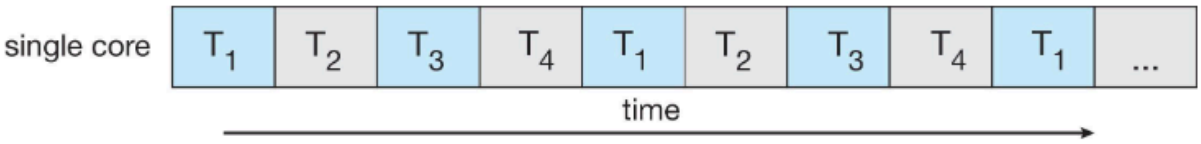


多核编程

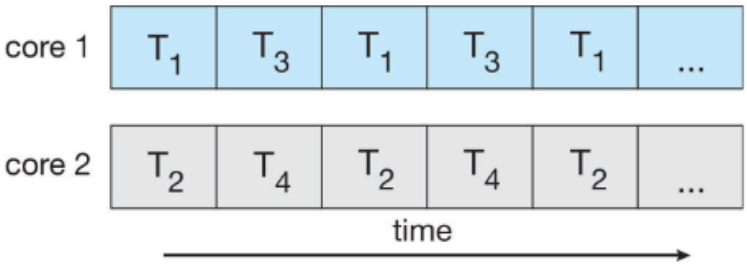
- 在多核或多处理器系统中，编程设计面临着重大的新挑战，包括：
  - 划分任务 – 如何划分为单独的并发任务 平衡 – 每个任务执行“相等”的工作量 数据拆分 – 任务使用的数据必须划分为在单独的核心上运行 数据依赖 – 如果存在数据依赖，则需要同步测试和调试——不同的执行路径使得调试变得困难
- 并行性和并发性之间有明显的区别 并行性 - 一个系统可以同时执行多个任务
- 并发性支持多个任务以取得进展
-
  - 随着时间的推移复用，单处理器核心，调度程序提供并发性
- 多核系统的出现需要一种全新的软件系统设计方法，特别强调并行编程。

并发与并行

- 单核系统上的并发执行——随时间复用

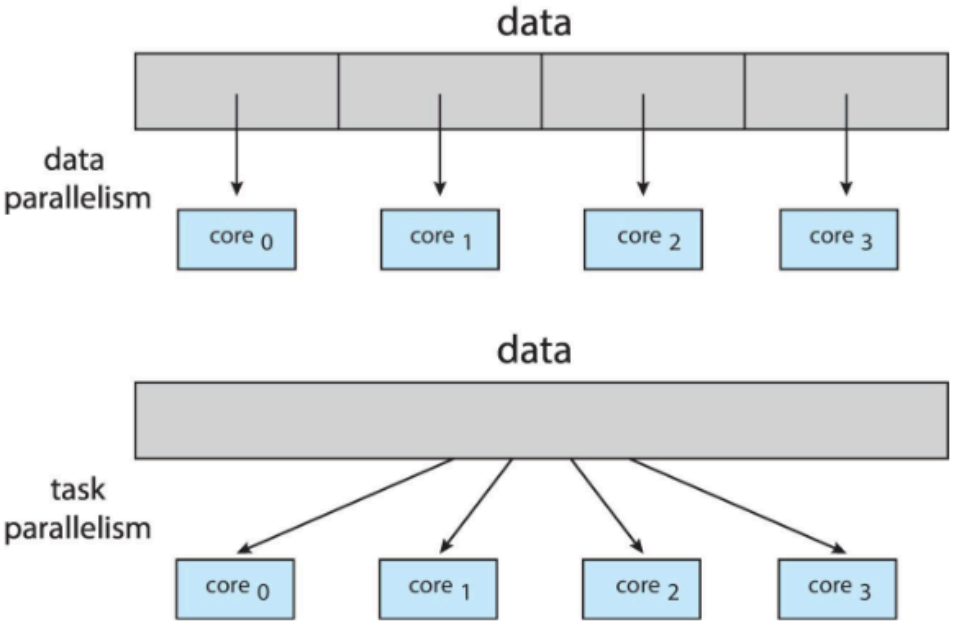


- 多核系统上的并行执行：



数据和任务并行

- 数据并行性 - 将数据子集分布在多个核心上，每个核心上执行相同的操作（在分布式机器学习任务中常见）
- 任务并行性 - 跨核心分配线程，每个线程执行独特的操作 数据和任务并行性并不相互排斥，应用程序可以同时使用两者 - 混合



阿姆达尔定律

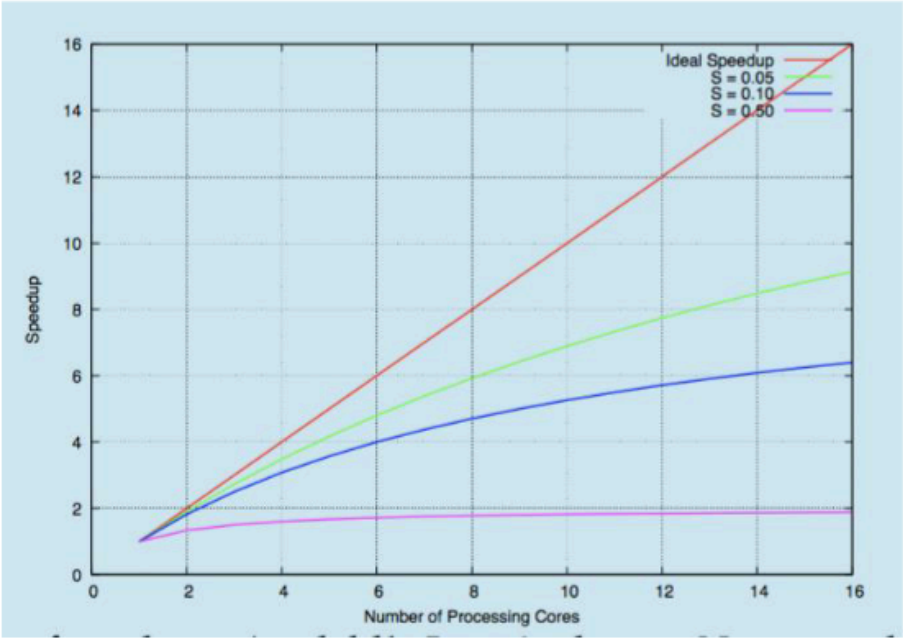
- 它通过向同时具有串行和并行组件（在程序中）S 是串行部分和 1-S 是并行部分的应用程序添加额外的内核来确定性能增益（在固定工作负载下执行任务的延迟的理论加速）
-
- N个处理核心

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- 也就是说，如果应用程序是 75% 并行和 25% 串行，则从 1 核变为 2 核会导致加速 1.6 倍
- 当 N 接近无穷大时，加速比接近 1/S

应用程序的串行部分对通过添加额外内核获得的性能产生不成比例的影响

阿姆达尔定律


Ideal Speedup
S = 0.05
S = 0.10
S = 0.50
Speedup
Number of Processing Cores
0
2
4
6
8
10
12
14
16

多线程进程

- 多线程进程有多个执行实例 - 每个实例都有一个程序计数器正在被获取和执行
- 线程类似于进程，只不过它们共享相同的地址空间，因此可以与进程中的其他线程访问同一组数据。单个线程的状态也类似于进程的状态 - 它具有程序计数器 (PC)，用于跟踪程序从何处获取指令。每个线程都有自己的私有执行寄存器集
- 如果两个线程在单个处理器上运行，则当从运行一个线程 (T1) 切换到运行另一个线程 (T2) 时，还必须进行上下文切换
  - 在运行 T2 之前，必须保存 T1 的寄存器状态并从 T2 的堆栈恢复 T2 的寄存器状态
  - 地址空间保持不变，上下文切换开销要小得多 当切换到属于不同进程的线程时，开销更多
- 这提供了进程（多线程）执行的并行性，并且可以实现 I/O 与单个程序中的其他活动的重叠
  - 一个线程正在 CPU 上运行，进程的另一个线程正在执行 I/O

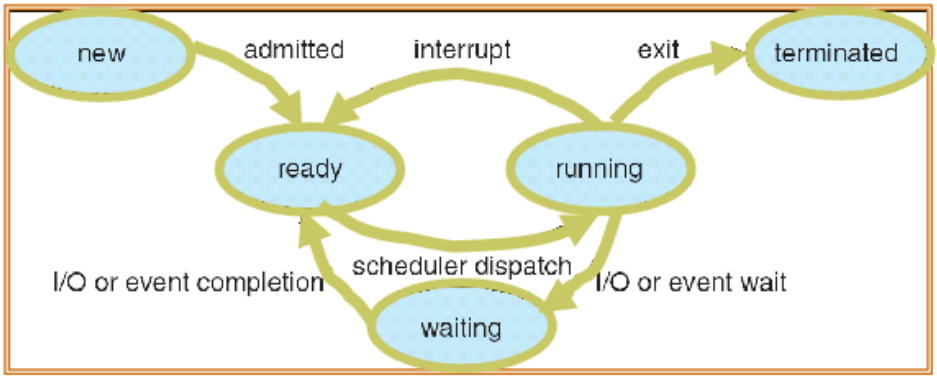
线

- 线程：单一唯一的执行上下文——轻量级进程
  - 程序计数器、寄存器、执行标志、堆栈 当线程驻留在寄存器中时，该线程正在处理器上执行。 PC寄存器保存线程中执行指令的地址寄存器保存线程的根状态（内存中的其他状态）
- 每个线程都有一个线程控制块（TCB）
  - 执行状态：CPU寄存器、程序计数器、堆栈指针调度信息：状态（稍后详细介绍）、优先级、CPU时间
  - 会计信息 各种指针（用于实现调度队列） 指向封闭进程的指针：PCB，它属于哪个进程

线程状态

- 线程在某种程度上封装了并发性，它是进程的“活动”组件
- 地址空间封装了保护：可以将其视为进程的“被动”部分
  - 一个程序的地址空间与另一程序的地址空间不同，以防止有错误的程序破坏整个系统
- 进程/地址空间中所有线程共享的状态
  - 内存内容（全局变量、堆） I/O 状态（文件描述符、
  - 网络连接等）
- 为每个线程声明“私有”
  - 保存在 TCB □ 线程控制块 CPU 寄存器（包括程序计
  - 数器） 执行堆栈（参数、临时变量、PC 保存）
  -

线程的生命周期



- 当线程执行时，它会改变状态：

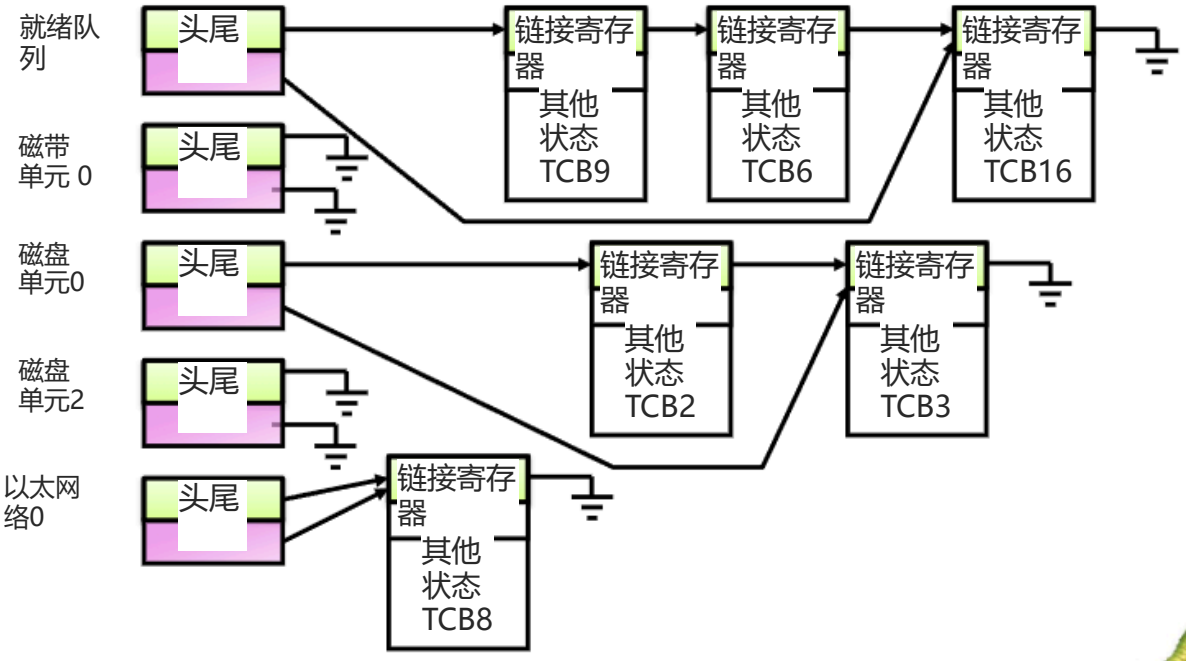
  new：正在创建线程 就绪：线程正在等待运行
  running：正在执行指令 waiting：线程等待某个事件发生 终止：线程已完成执行 “活动”线程由其 TCB 表示

-
  - TCB 根据其状态组织成队列

就绪队列和各种I/O设备队列

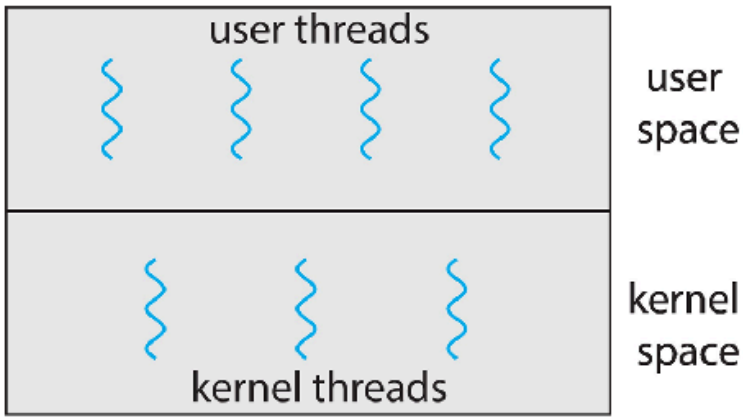
线程未运行 □ TCB 位于其他队列中

- 每个设备/信号/条件的队列，每个都有自己的调度策略



用户线程和内核线程

- 用户线程——程序中独立可执行的实体，由用户级线程库创建和管理
- 内核线程 – 可以在 CPU 上调度和执行，由操作系统支持和管理
- 示例 – 几乎所有通用操作系统，包括：Windows、Linux、Mac OS X iOS、Android



多线程模型

- 用户级线程仅对程序员可见，而对内核未知。因此，它们不能被安排在 CPU 上运行。换句话说，操作系统只管理和调度内核线程，线程库提供了用于在程序中创建和管理用户线程的API。主要的线程库有 POSIX Pthreads、Windows 线程和 Java 线程三种，分别对应三种最常见的 API，例如 Win32 API、POSIX API 和 Java API
- 在某种程度上，用户级线程在程序中提供了操作系统可以利用的并发和模块化实体。或者，如果程序中没有定义或指定用户线程，则不会有可以调度程序执行的内核线程

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);
```

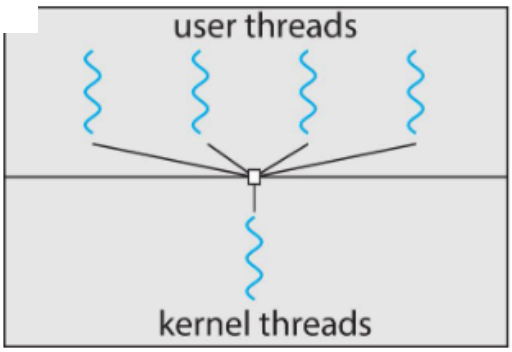
多线程模型（续）

- 最终，用户线程和内核线程之间必须存在映射。建立这种关系的常用方法有以下三种：
  - 多对一：许多用户级线程映射到一个内核线程 一对一：每个用户级线程映射到一个内核线程 – 在现代操作系统中最常见
  - 多对多：许多用户级线程映射到许多内核（通常数量较少）线程

多对一

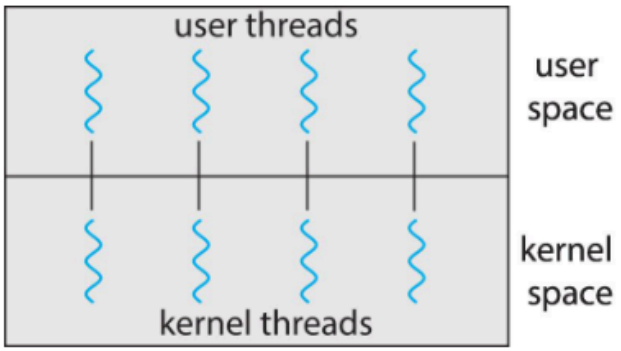
- 许多用户级线程映射到单个内核线程 当前哪个用户线程映射到内核线
- 程是一个调度问题，称为进程争用范围或 PCS（将在第 5 章中讨论）

- 一个线程阻塞（即内核线程）会导致映射到该线程的所有线程都阻塞，即整个进程被阻塞
- 多个用户线程无法在多核系统上并行运行，因为一次只能有一个内核线程处于活动状态
- 目前很少有系统使用此模型示
- 例：
  - Solaris 绿色线程 GNU
  - 可移植线程



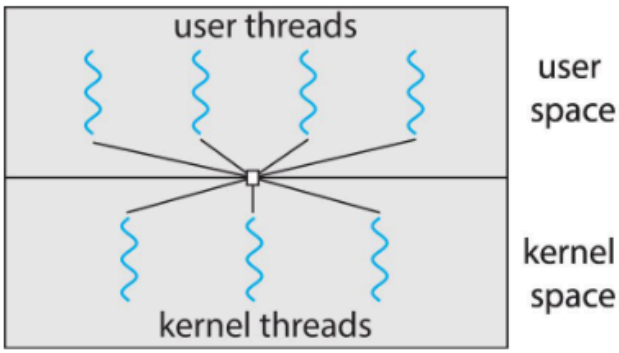
一对一

- 每个用户级线程映射到一个内核线程，它提供了最大的并发性，并且还允许多个线程在多处理器或多核系统上并行运行
- 这意味着创建用户级线程需要创建相应的内核线程——每个进程的线程数量有时会由于开销而受到限制（内核线程消耗内存、I/O 等资源）
- 示例
  - 视窗
  - Linux



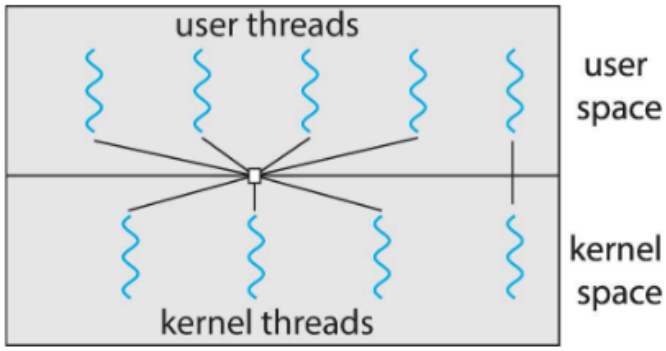
多对多模型

- 将许多用户级线程多路复用为更少或相同数量的内核线程，特定于特定应用程序或机器这为进程执行提供了一定程度的并发性 - 调度还涉及进程争用范围或 PCS
- 这使得操作系统可以提前创建足够数量的内核线程——线程池
- 带有 ThreadFiber 包的 Windows 否则不太常见



两级模型

- 与多对多模型类似，不同之处在于它允许用户线程绑定到内核线程



多线程映射模型

- 多对多模型最灵活，但在实践中可能很难实现
  - 例如，在具有八个处理核心的系统上，应用程序可能会比在具有四个核心的系统上分配更多的内核线程
- 随着现代计算机系统中处理核心数量的不断增加，限制内核线程的数量变得不再那么重要。现在大多数操作系统都使用一对一模型
-

线程问题

- fork() 和 exec() 系统调用的语义
- 信号处理

  目标线程的同步和异步线程取消
-
  - 异步或延迟取消

fork() 和 exec() 的语义

- 线程调用的 fork() 是仅复制调用线程还是进程的所有线程?
- 一些 UNIX 系统选择使用两个版本的 fork() - 如果在 fork 后立即调用 exec()，则无需复制所有线程，因为 exec() 中指定的程序将替换整个进程。在这种情况下，仅复制调用线程是合适的。
- exec() 通常正常工作 – 替换整个进程，包括所有线程

信号处理

- 信号在 UNIX 系统中用于通知进程某个特定的信息
  事件已经发生。信号可以同步或异步接收，具体取决于信号源和原因
  - 同步信号包括非法内存访问和除以 0 - 传递到执行导致该信号的操作的同一进程（这就是它们被认为是同步的原因）。当信号由正在运行的进程外部的事件生成时 - 异步信号。示例包括使用特定击键终止进程（例如<control><C>）并且计时器到期
- 信号以不同的方式处理
  - 某些信号可能会被忽略，而其他信号（例如非法内存访问）则通过终止程序来处理

信号处理（续）

- 信号处理程序用于处理信号，遵循以下模式
  - 信号由特定事件（例如进程终止）生成 信号被传递到进程
  -
  - 信号由两个信号处理程序之一处理，默认或用户定义
- 每个信号都有内核用来处理该信号的默认处理程序
  - 用户定义的信号处理程序可以覆盖默认处理程序
- 对于单线程进程，信号被传递到该进程或线程
- 多线程进程的信号应该传递到哪里？
  - 将信号传递给应用该信号的线程 将信号传递给进程中的每个线程
  -
  - 将信号传递给进程中的某些线程分配特定线程来接收进程的所有信号
  -

信号处理（续）

- 传递信号的方法取决于生成的信号类型
  - 例如，同步信号需要传递给引发该信号的线程，而不是传递给进程中的其他线程
  - 一些异步信号——例如终止进程的信号（<control><C>，例如）—应该发送到所有线程
- 用于传递信号的UNIX函数kill(pid t pid, int signal)，它指定特定信号（signal）要传递到的进程（pid）
- POSIX Pthreads 函数允许将信号传递到指定线程 (tid)：pthread_kill(pthread t tid, int signal)

线程取消

- 在线程完成之前将其终止，要取消的线程称为目标线程。
- 取消发生在两种不同的情况下
  - 异步取消立即终止目标线程延迟取消目标线程定期检查是否应该终止，使其有机会以有序的方式终止自身
  -
- 调用线程取消请求取消，但实际取消取决于线程状态和类型
- 在 Pthreads 中，如果线程禁用了取消，则取消将保持挂起状态，直到线程启用它 - 默认类型是延迟的

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

线程取消（续）

- 创建和取消线程的 Pthread 代码：
  - Pthreads：线程编程的 POSIX 标准 需要
  - #include<pthread.h>

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

  . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

开始例行公事

操作系统示例

- Windows 线程
- Linux 线程

Windows 线程

- Windows应用程序作为一个单独的进程运行，每个进程可能包含一个或多个线程
- Windows API – Windows 应用程序的主要 API Windows 使用
- 一对一映射，其中每个用户级线程映射到关联的内核线程

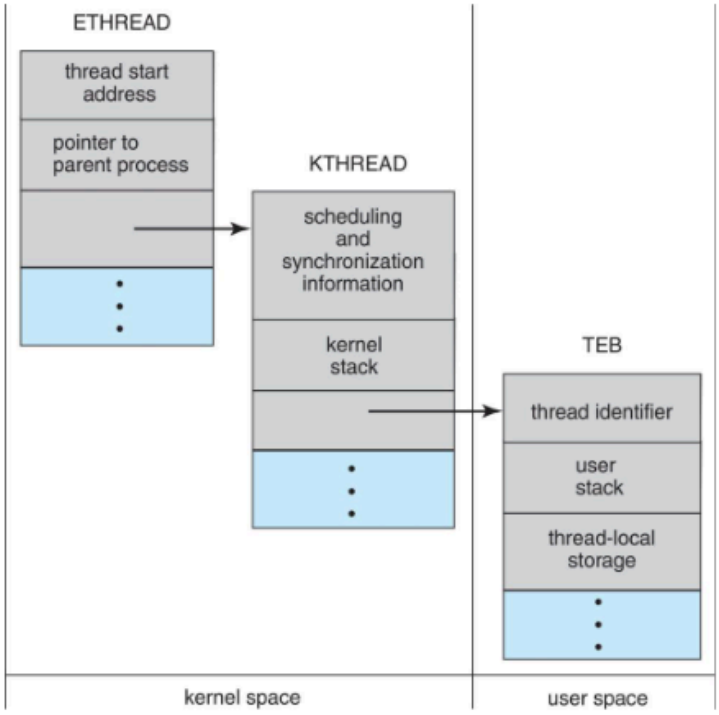
- Window 线程的一般组件包括：
  - 标识线程的线程 ID 代表处理器状态的寄存
  - 器组 程序计数器 (PC)
  -
  - 当线程在用户模式或内核模式下运行时，将用户堆栈和内核堆栈分开运行
  - 时库和动态链接库（DLL）使用的私有数据存储区域

- 寄存器组、堆栈和私有存储区域称为线程的上下文

Windows 线程（续）

线程的主要数据结构包括：

- ETHREAD（执行线程块）——包括指向线程所属进程和 KTHREAD 的指针，
- KTHREAD（内核线程块）调度和同步信息，内核模式堆栈，指向 TEB 的指针，在内核空间中 TEB（线程环境块）– 线程 id，用户模式堆栈，线程本地存储，在用户空间中



Linux 线程

- Linux 不区分线程和进程。对于Linux来说，线程只是一种特殊的进程，统称为任务
- Linux 中的线程是可以与其他线程共享某些资源的进程。每个线程都有一个唯一的task_struct（在第3章中说明），并显示为一个普通进程——Linux中的线程恰好与其他进程共享资源，例如地址空间。这种线程方法与Microsoft Windows等操作系统形成鲜明对比或 Sun Solaris，它具有对线程的显式内核支持（有时将线程称为轻量级进程）
- 例如，如果一个进程由两个线程组成
  - 在 Microsoft Windows 上，存在一个 PCB 来描述共享资源（例如地址空间或打开的文件），并且依次指向两个不同的线程。每个线程TCB描述它单独拥有的资源。在Linux中，只有两个具有正常task_struct结构的任务。两个进程设置为共享资源

Linux 线程

- 线程的创建方式与普通任务相同，不同之处在于clone()系统调用用于传递与要共享的特定资源相对应的标志：

```
克隆（CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND，0）；
```

  - clone() 允许子任务与父任务共享执行上下文的一部分，例如地址空间、文件描述符、信号处理程序
  - 标志控制行为

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- 新任务不是像 fork() 那样复制所有数据结构，而是根据传递给 clone() 的标志集指向父任务的数据结构。普通的 fork() 可以实现为 clone(SIGCHLD,0);

第 4 章结束