OPERATING SYSTEMS

# Threads

## Operating Systems

郑贵锋 博士
中山大学计算机学院
zhenggf@mail.sysu.edu.cn
https://gitee.com/code_sysu

中山大學 SUN YAT-SEN UNIVERSITY

## Overview



- **目录**
  - **概述**
  - **多核编程**
  - **用户和内核级线程**
  - **多线程模型**
  - 线程库
    - POSIX线程Pthreads
  - 隐式线程化实现
    - 线程池
    - OpenMP
  - 线程的问题
  - Linux clone()
  - 示例：Windows线程

## Overview



- **进程特性回顾**
  - 资源所有权单位
    - 为进程分配一个地址空间来保存进程映像。
      - 控制某些资源（文件、I/O设备...）
  - 调度单位
    - 进程是通往一个或多个程序的执行路径：
      - 执行可以与其他进程交错。
      - 进程具有执行状态和调度优先级。
  - 这些特征单独处理：
    - 资源所有权的单位通常被称为任务或（出于历史原因）也被称为进程。
    - CPU调度单元通常指线程(Thread)或轻量级进程(LWP)
      - 传统的重量级进程(HWP)相当于一个具有单个线程的任务
  - 多线程-同一任务中可以存在多个线程
    - 多线程是操作系统在单个进程中支持多个并发执行路径的能力

## Overview



- **线程的动机**
  - 大多数现代应用程序都是多线程的。
    - 线程在一个应用程序中运行。
  - 一个应用程序的多个任务可以由分离的线程实现。例如：
    - 更新显示
    - 获取数据
    - 拼写检查
    - 回答网络请求
  - 进程创建很重量级，而线程创建很轻量级。
  - 使用线程可以简化代码，提高效率。
  - 内核通常是多线程的

## Overview



- **任务/进程 vs. 线程**
  - 每个任务/进程的项目（由任务的所有线程共享）：
    - 保存进程映像的地址空间
    - 全局变量
    - 对文件、I/O和其他资源的受保护访问
    - 子进程
    - 待处理警报
    - 信号和信号处理程序
    - 会计信息
  - 每个线程的项目：
    - 执行上下文/状态（正在运行、准备就绪等）
      - 未处于运行状态时保存
    - 程序计数器
    - 寄存器集
    - 执行堆栈
    - 每个线程的局部变量的静态存储 (TLS)

## Overview

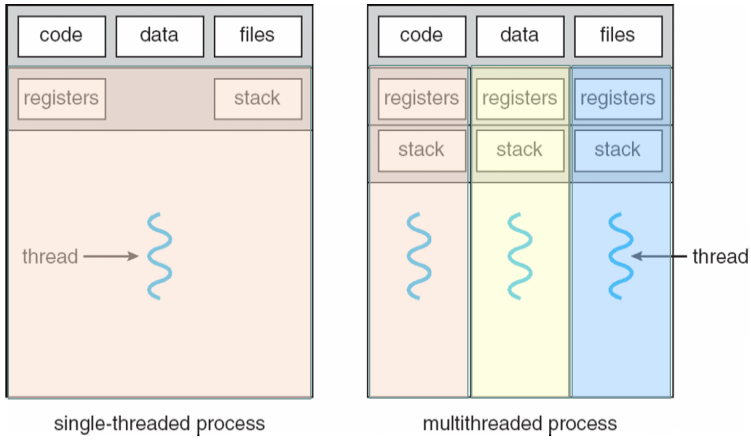


- **任务/进程 vs. 线程**
  - 线程可以访问其任务的地址空间和资源：
    - 内存地址空间和资源由任务的所有线程共享。
    - 当一个线程更改（非私有）内存项时，任务的所有其他线程都可以看到这一点。
    - 使用一个线程打开的文件可供任务的其他线程使用。

## Overview



- **任务/进程 vs. 线程**

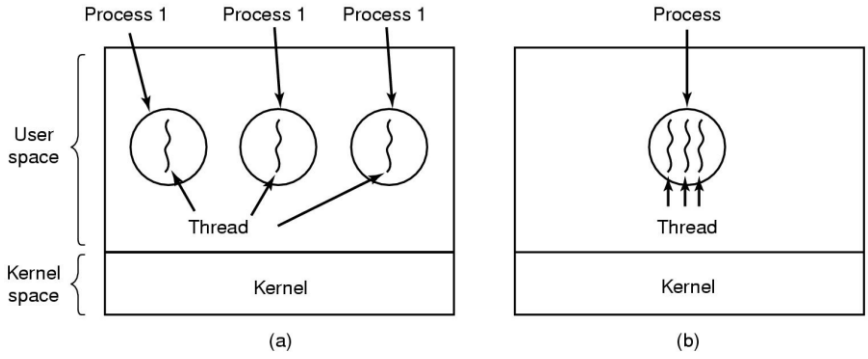



- 单线程和多线程进程

## Overview



- **任务/进程 vs. 线程**

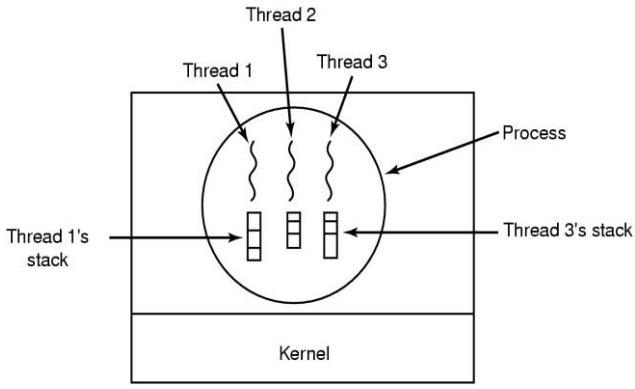



(a) 三个进程，每个进程有一个线程
(b) 一个进程有三个线程

## Overview



- **任务/进程 vs. 线程**

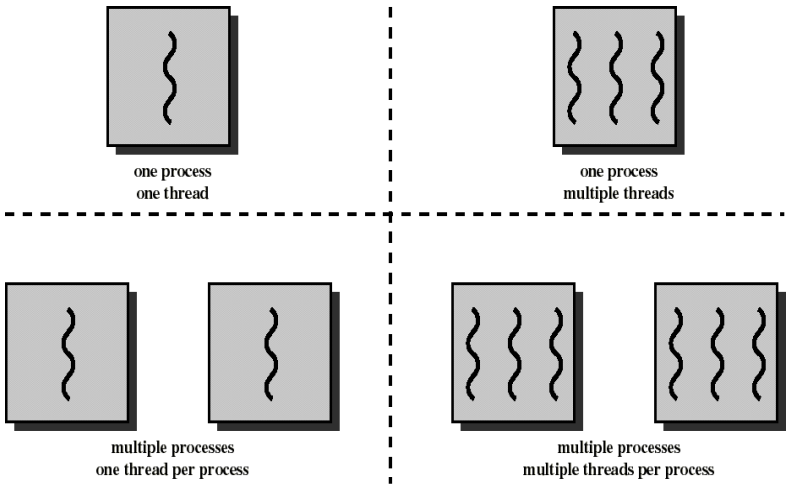



每个线程都有自己的堆栈

## Overview



- **任务/进程 vs. 线程**




线程和进程的组合

## Overview



- **任务/进程 vs. 线程**
  - 创建和管理进程通常被认为是一项昂贵的任务。
    - 例如，fork()系统调用。
  - 确保所有进程在系统上和平共存并不容易。
    - 因为并发透明是有代价的
  - 与其让操作系统负责并发透明性，不如让单个应用程序来管理每个线程的创建和调度。
    - 线程可以被认为是"程序的一部分（在用户空间中）的执行"。

## Overview



- **任务/进程 vs. 线程**
  - 线程和进程之间的关系
    - 一对一，多对一，多对多

| 线程：进程 | 描述 | 示例系统 |
|---|---|---|
| 1:1 | 每个执行线程都是一个具有自己地址空间和资源的唯一进程。 | 传统的UNIX实现 |
| M:1 | 进程定义地址空间和动态资源所有权。可以在该进程中创建和执行多个线程。 | Windows NT、Solaris、Linux、OS/2、OS/390、MACH（CMU，1985） |
| 1:M | 线程可以从一个进程环境迁移到另一个进程环境。这允许线程在不同的系统之间轻松移动。 | Ra（Clouds，1989），Emeralds（嵌入式实时分布式系统可扩展微内核，1999） |
| M:N | 组合M:1和1:M案例的属性。 | TRIX（MIT，1970年） |

## Overview



- **线程的好处**
  - 多线程编程的好处可分为四大类：
    - 响应能力
      - 如果进程的部分被阻塞，多线程可允许继续执行，这对于用户界面交互尤其重要。
        - 当一个服务器线程被阻塞并等待时，同一任务中的另一个线程可以运行。
    - 资源共享
      - 线程共享进程资源，比IPC（如共享内存或消息传递）更容易实现。
        - 线程间通信和同步速度更快。

## Overview

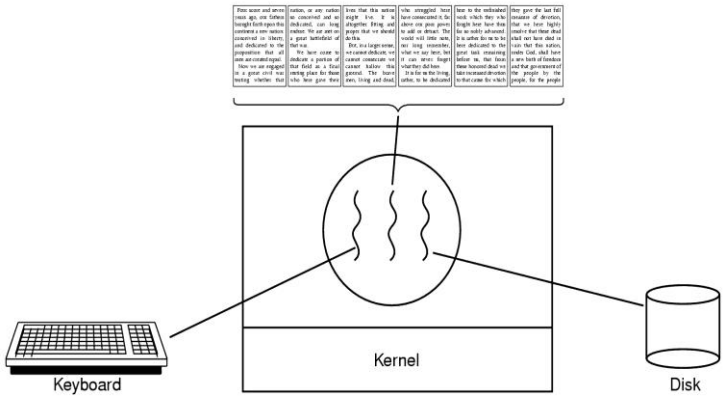


- **线程的好处**
  - 多线程编程的好处可以分为四大类：（续）
    - 经济性/低系统开销
      - 线程创建和上下文切换的开销比进程低。
        - 创建和终止线程的时间少于进程，因为我们不需要另一个地址空间
        - 在两个线程之间切换的时间比在进程之间切换的时间少，因为我们不需要处理地址空间。
    - 可扩展性/可伸缩性
      - 线程可以在不同的处理内核上并行运行，可利用多处理器/多核体系结构和网络/分布式系统

## Overview



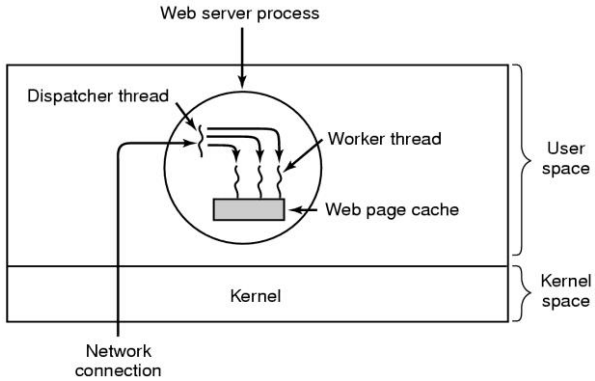
- **线程的好处**
  - 示例1：字处理器
    - 一个线程显示菜单并读取用户输入，而另一个线程执行用户命令，第三个线程写入磁盘。
      - 应用程序的响应速度更快。




## Overview



- **线程的好处**
  - 示例2：文件/Web服务器
    - 服务器需要在短时间内处理多个文件/页面请求。因此，为每个请求创建（并销毁）单个线程更有效。
    - 在SMP（对称多处理器）机器上：多个线程可能在不同的处理器上同时执行。



## Overview



- **线程的好处**
  - 示例2：文件/Web服务器
    - 调度程序线程

```
while（TRUE）{
    获取下一个请求（&buf）；
    交接工作（&buf）；
}
```

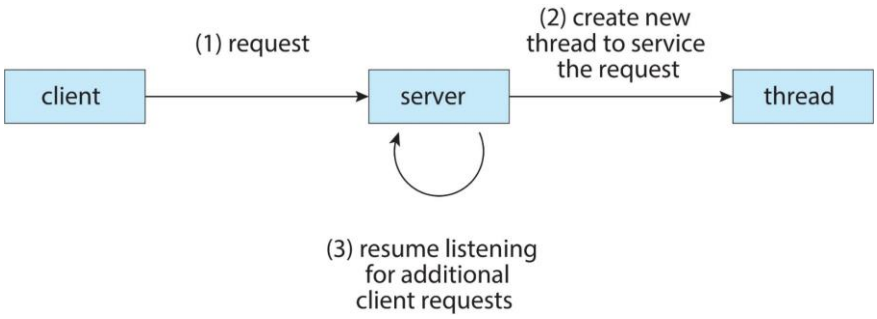
    - 工作线程

```
while（TRUE）{
    等待工作（&buf）；
    在缓存中查找页面（&buf，&page）；
    如果（页面不在缓存中（&page））
        从磁盘读取页面（&buf，&page）；
    返回页面（&page）；
}
```

## Overview



- **线程的好处**
  - 多线程服务器体系结构



## Overview



- **线程的好处**
  - 两个重要的含义
    - 线程化应用程序通常比非线程化应用程序运行得更快。
      - 因为避免了内核和用户空间之间的上下文切换
    - 线程应用程序更难开发。
      - 虽然简单、干净的设计有所帮助
  - 此外，假设开发环境为开发人员提供了一个线程库。
    - 大多数现代环境都是这样

## Overview



- **应用线程的好处**
  - 考虑一个由几个独立的部分组成的应用程序，它们不需要按顺序运行。
    - 每个部分都可以实现为一个线程。
  - 每当一个线程被阻塞等待I/O时，执行可能会切换到同一应用程序的另一个线程。
    - 而不是阻塞它并切换到另一个应用程序
  - 由于同一进程中的线程共享内存和文件，因此它们可以在不调用内核的情况下相互通信。
  - 并发要求
    - 有必要同步各个线程的活动，以便它们不会获得当前数据的不一致视图。

## Overview



- **线程状态**
  - 线程有三种关键状态。
    - 运行状态
    - 就绪状态
    - 阻塞状态
  - 它们没有挂起(suspend)状态，因为同一任务中的所有线程共享相同的地址空间。
    - 实际上，挂起（例如交换）单个线程涉及挂起同一任务的所有线程。
  - 任务的终止将终止任务中的所有线程。

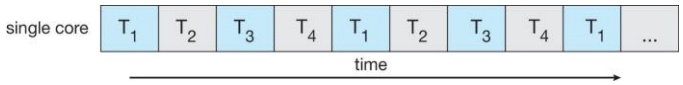
## Multicore Programming



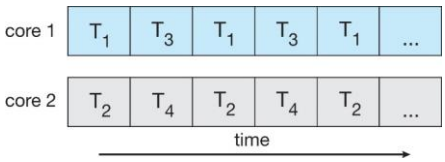
- **多核系统**
  - 多核体系结构将多个计算核放在一个处理芯片上。
    - 对于操作系统来说，每个内核都是一个独立的处理器。
    - 无论核心出现在CPU芯片之间还是CPU芯片内部，我们称这些系统为多核或多处理器系统。
  - 多线程编程为更有效地使用这些多计算核心提供了一种机制，并改善了并发性。
  - *并行Parallelism*与*并发Concurency*
    - 并行系统可以同时执行多个任务。
    - 并发系统允许所有任务取得进展，从而支持多个任务。因此，可以在没有并行性的情况下实现并发。
      - 在SMP和多核体系结构出现之前，大多数计算机系统只有一个处理器。CPU调度器被设计为通过在系统中的进程之间快速切换来提供并行性的假象，从而允许每个进程取得进展。这些进程是并发运行的，但不是并行运行的。

## 多核系统

- 在具有多个核心的系统上，并发意味着线程可以并行运行，因为系统可以为每个核心分配单独的线程。
  - 示例：线程T1、T2、T3和T4在单核系统上并发执行。




  - 示例：线程T1、T2、T3和T4在2核系统上并行执行。



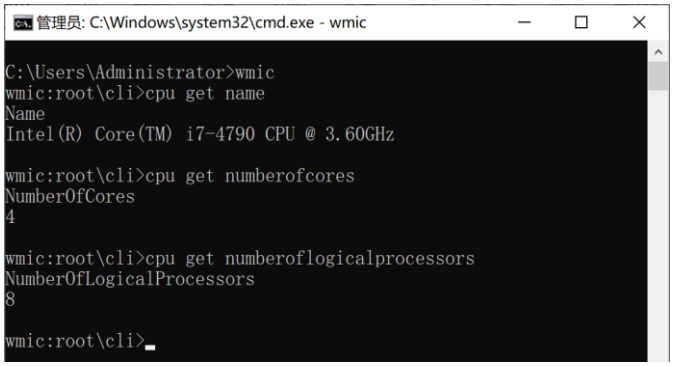



## 多线程

- 随着系统从数十个线程发展到数千个线程，CPU设计者通过添加硬件来提高线程性能，从而提高了系统性能。
  - 现代Intel CPU通常支持每个内核两个线程（作为逻辑处理器）
  - Oracle T4 CPU支持每个核心八个线程。这种支持意味着可以将多个线程加载到内核中以实现快速切换。
- 多核计算机无疑将继续增加核心数量和硬件线程支持。

## 多线程

- 实例
  - 在Windows命令行界面（CLI）中，查询CPU信息。




  - NumberOfLogicalProcessors是Intel i7-4790 CPU支持的最大线程数，每个物理内核支持两个线程（逻辑处理器）。



## 多线程

- 编程挑战
  - 对于操作系统设计者
    - 实现系统模型以支持具有多个处理核心的并行执行
  - 对于应用程序程序员
    - 修改或设计多线程程序
  - 一般来说，在多核系统的编程中有五个方面存在挑战：

(1) 确定任务

- 检查应用程序，找到可以划分为独立、并发任务的区域，理想情况下，这些任务彼此独立，因此可以在单个核心上并行运行

(2) 均衡

- 确保确定的任务执行同等价值的同等工作

## 多线程

- 编程挑战
  - 一般来说，在多核系统的编程中有五个方面存在挑战：（续）

(3) 数据分割

- 由单独任务访问和操作的数据必须划分到单独的内核上运行。

(4) 数据依赖性

- 检查两个或多个任务之间的依赖关系。同步用于满足数据依赖关系

(5) 测试和调试

- 测试和调试在多个核上并行运行的并发程序非常困难，因为可能的执行路径太多了。



## 多线程

- 并行类型
  - 一般来说，并行有两种类型，包括数据并行和任务并行。
  - 数据并行性
    - 重点是在多个计算核心之间分布相同数据的子集，并在每个核心上执行相同的操作
  - 任务并行性
    - 涉及跨多个计算核心分发任务（线程），而不是数据。每个线程都在执行一个唯一的操作。不同的线程可能在相同的数据上操作，也可能在不同的数据上操作

## 多线程

- 多线程是一种编程类型，它利用CPU的能力跨多个处理核心同时处理多个线程。
  - 同时执行任务或指令
- 默认情况下，主线程在程序开始时运行，并创建新线程来处理任务
  - 这些新线程彼此并行运行，并且通常在完成后将其结果与主线程同步。



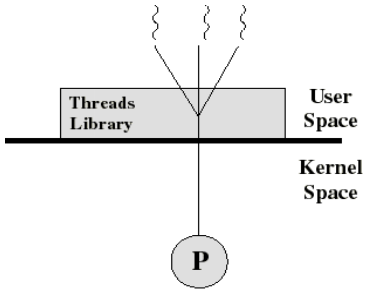
## 多线程级别

- 线程库包含代码并提供以下API：
  - 创建和销毁线程
  - 在线程之间传递消息和数据
  - 调度线程执行
  - 保存和恢复线程上下文
- 三个主线程库：
  - POSIX线程
  - Win32线程
  - Java线程
- 有三个多线程级别：
  - 用户级线程（ULT）
    - 线程库完全在用户空间中
  - 内核级线程（KLT）
    - 操作系统支持的内核级线程库
  - 混合ULT/KLT方法

## User and Kernel Level Threads



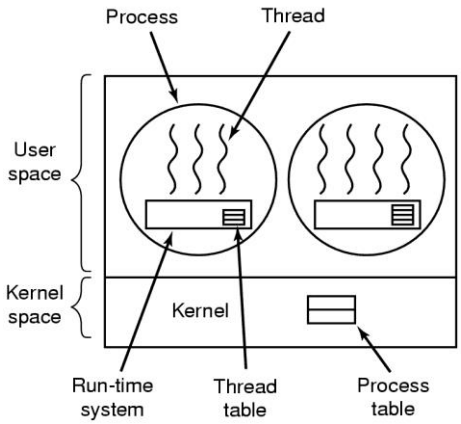
- **用户级线程（ULTs）**
  - ULTs的管理由具有用户级线程库的应用程序完成。
  - 内核不知道ULTs的存在。
  - 线程切换不需要内核模式特权。
  - 调度是特定于应用程序的。




## User and Kernel Level Threads



- **用户级线程（ULTs）**
  - 在用户空间中实现ULTs。



## User and Kernel Level Threads



- **用户级线程（ULTs）**
  - ULTs的内核活动
    - 内核不知道线程活动，但仍在管理进程活动。
    - 线程状态独立于进程状态。
      - 当线程进行系统调用时，整个任务将被阻塞。
      - 但对于用户级线程库，该线程仍处于运行状态。

## User and Kernel Level Threads

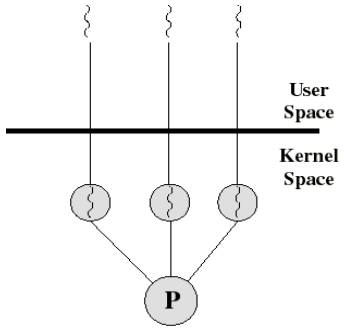


- **用户级线程（ULTs）**
  - ULTs的优缺点
    - 优势
      - 线程切换不涉及内核。
        - 无双模切换
      - 调度可以是特定于应用程序的。
        - 选择最佳算法
      - ULTs可以在任何操作系统上运行。
        - 只需要支持用户级线程库
    - 不便之处
      - 一个线程被阻塞可能会导致内核阻塞进程。
        - 进程中的所有线程都将被阻塞
      - 内核只能将整个进程分配给处理器
        - 例如，同一进程中的两个线程不能在两个处理器上同时运行

## User and Kernel Level Threads



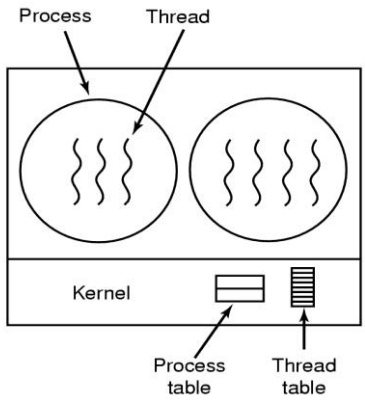
- **内核级线程（KLT）**
  - KLT的管理由内核完成。
  - 没有线程库，但为内核线程部件的用户提供了一个API。
  - 内核维护进程和线程的上下文信息。
  - 在线程之间切换需要内核。
  - 基于线程的调度




## User and Kernel Level Threads



- **内核级线程（KLT）**
  - 在内核中实现KLT



## User and Kernel Level Threads



- **内核级线程（KLT）**
  - KLT几乎包含在所有通用操作系统中。
    - Windows
    - OS/2
    - Linux
    - Solaris
    - Tru64 UNIX
    - MacOSX
  - Linux线程
    - Linux将线程看作任务
    - 线程创建是通过clone()系统调用完成的。
      - clone()允许子任务共享父任务（进程）的地址空间。
      - 这种地址空间的共享使得克隆子任务的行为非常类似于一个单独的线程。

## User and Kernel Level Threads



- **内核级线程（KLT）**
  - KLT的优势和不足
    - 优势
      - 内核可以在多个处理器上同时调度同一进程的多个线程。
      - 阻塞是在线程级别上完成的。
      - 内核例程可以是多线程的。
    - 不足之处
      - 同一进程内的线程切换涉及内核。
        - 这将导致显著的减速。

| Operation | User-Level Threads | Kernel-Level Threads | Processes |
|---|---|---|---|
| Null Fork | 34 | 948 | 11,300 |
| Signal Wait | 37 | 441 | 1,840 |

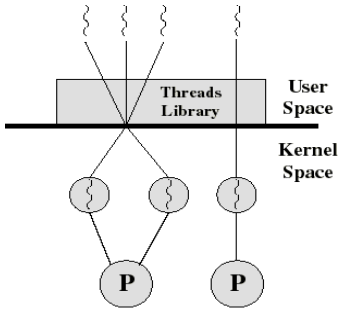
线程操作延迟($\mu s$)

(*Anderson, T.* et al, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", ACM TOCS, February 1992.)

## User and Kernel Level Threads



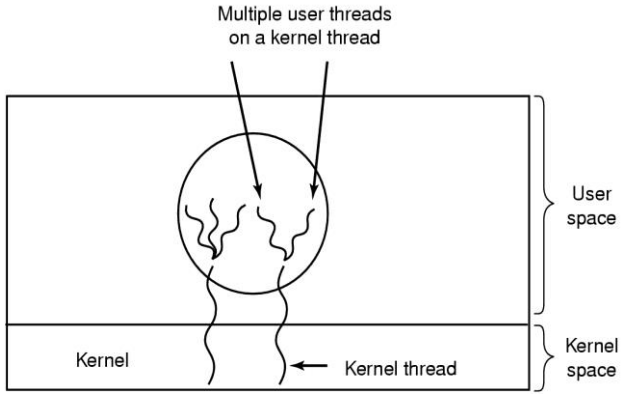
- **混合ULT/KLT方法**
  - 线程创建是在用户空间中完成的
  - 线程的大量调度和同步是在用户空间中完成的
  - 程序员可调整KLT的数量
  - 可以结合两种方法的优点
  - 示例：版本9之前的Solaris




## User and Kernel Level Threads



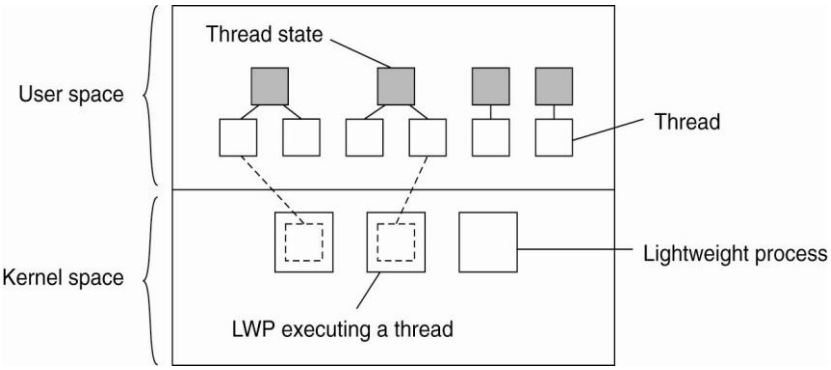
- **混合ULT/KLT方法**
  - 混合实现



## User and Kernel Level Threads



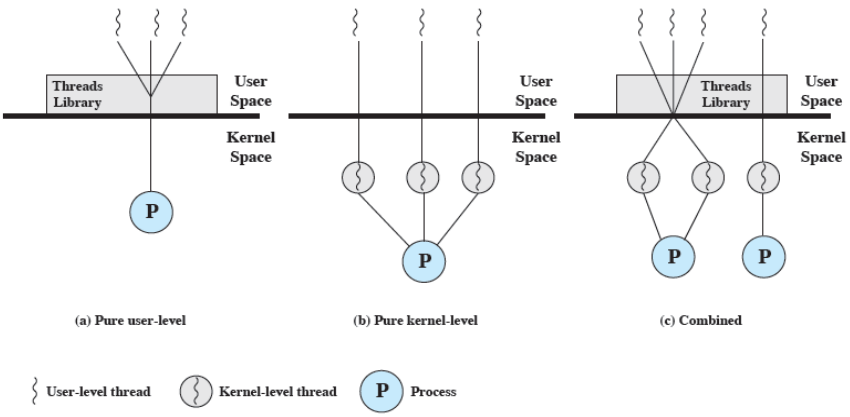
- **混合ULT/KLT方法**
  - 混合实现




## User and Kernel Level Threads



- **混合ULT/KLT方法**
  - ULT、KLT和组合方法。



## Multithreading Models

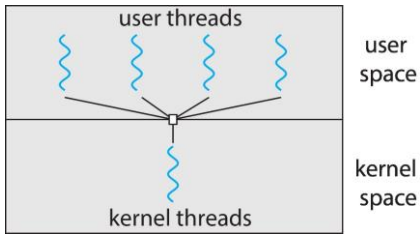


- **多线程模型**
  - 对线程的支持可以在用户级别提供用户线程（ULT），也可以由内核提供内核线程（KLT）。
    - 用户线程在内核之上受支持，并且在没有内核支持的情况下进行管理，而内核线程则由操作系统直接支持和管理。
  - 用户线程和内核线程之间的关系是
    - 多对一模型
    - 一对一模型
    - 多对多模型

## Multithreading Models



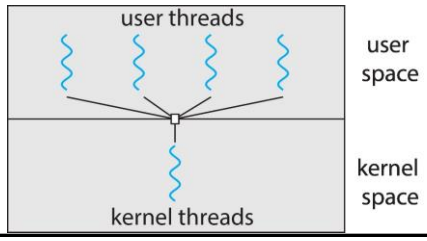
- **多线程模型**
  - 多对一模型
    - 许多用户级线程映射到单个内核级线程



## Multithreading Models



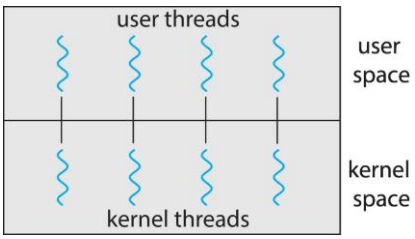
- **多线程模型**
  - 多对一模型
    - 线程由用户空间中的线程库有效地管理。
    - 一个线程阻塞会导致进程（及其所有线程）阻塞。
    - 多线程不能在多核系统上并行运行，因为一个内核中只能有一个线程。
      - 目前很少有系统使用这种模式。
    - 例子
      - Solaris绿色线程
      - GNU可移植线程




## Multithreading Models



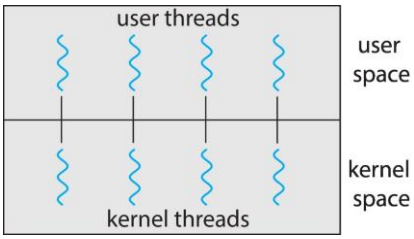
- **多线程模型**
  - 一对一模式
    - 每个用户级线程映射到一个内核级线程。



## Multithreading Models



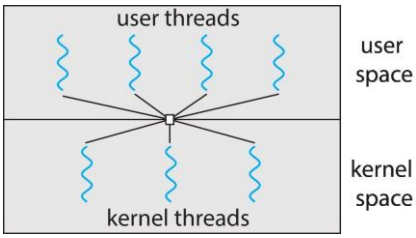
- **多线程模型**
  - 一对一模式
    - 创建用户级线程将创建内核线程。
    - 比多对一更具并发性。
      - 当线程阻塞时允许另一个线程运行
      - 允许多线程在多处理器上并行运行
    - 由于创建内核线程的开销，每个进程的线程数有时会受限，这可能会给系统性能带来负担。
    - 例子
      - Windows
      - *Linux*
      - Solaris 9及更高版本




## Multithreading Models



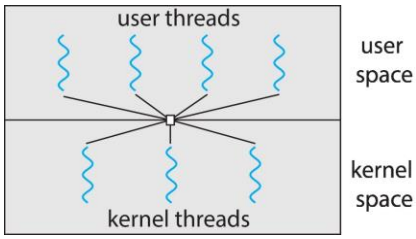
- **多线程模型**
  - 多对多模型
    - 将多个用户级线程多路复用到更小或相等数量的内核线程。



## Multithreading Models



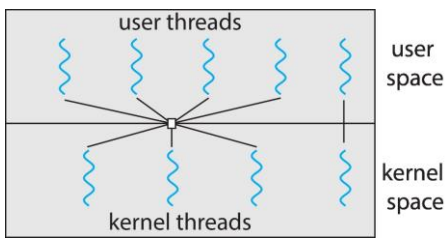
- **多线程模型**
  - 多对多模型
    - 开发人员可以根据需要创建足够数量的用户线程，相应的内核线程可以在多处理器上并行运行。
    - 当一个线程阻塞时，内核可以安排另一个线程执行。
    - 例子
      - Solaris版本9之前的版本。
      - 使用*ThreadFiber*软件包的Windows




## Multithreading Models



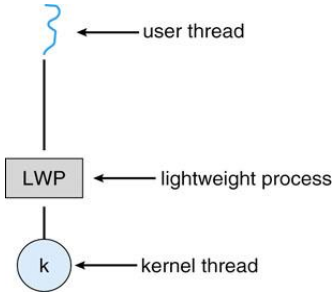
- **多线程模型**
  - 双层模型
    - 类似于多对多模型，只是它允许用户线程绑定到内核线程。
    - 例子
      - IRIX（由SGI提供的图形工作站）
      - HP-UX（惠普9000系列服务器的操作系统）
      - Tru64 UNIX
      - Solaris 8及更早版本。



## Multithreading Models



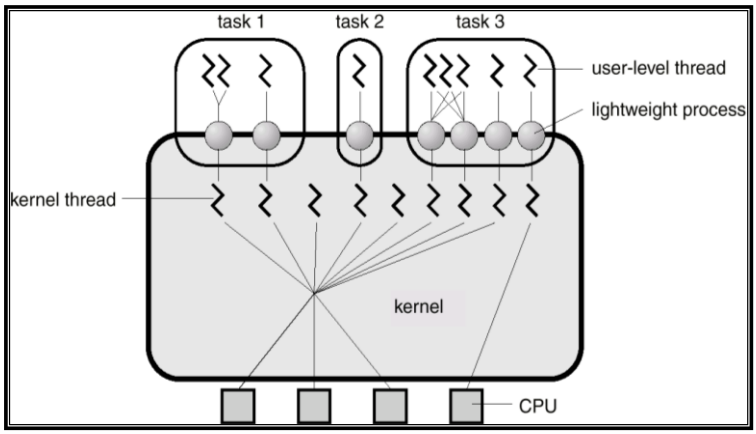
- **多线程模型**
  - 轻量级进程（LWP）
    - 轻量级：在大多数系统中，LWP与普通进程的区别在于它只有一个最小的执行上下文和调度程序所需的统计信息
    - 重要作用：提供了一个用户级线程实现的中间系统，通过系统调用获得内核提供的服务。当一个用户级线程运行时，只需要将它连接到一个LWP上便可以具有内核支持线程的所有属性。




## Multithreading Models



- **多线程模型**
  - 示例：Solaris 2线程



## Multithreading Models



- **多线程模型**
  - 示例：Java线程
    - Java线程由JVM管理。
    - 通常使用底层操作系统提供的线程模型实现。
    - Java线程可以通过以下方式创建：
      - 继承Thread类（在语言级别）
      - 实现Runnable接口。