



中国科学院大学
University of Chinese Academy of Sciences

操作系统课程 B0911010Y-01

线程

中国科学院大学计算机与控制学院

中国科学院计算技术研究所

2019-09-11





内容提要

- 线程的概念
- 线程表示与操作API
- 线程模型



一个例子

- 编写一个MP3播放软件。核心功能模块有三个：
 - 从MP3音频文件中读取数据
 - 对数据进行解压缩
 - 把解压缩后的音频数据播放出来

```
main( )
{
    while(TRUE )
    {
        I/O ——— Read( );
        CPU ——— Decompress( );
                Play( );
    }
}
Read( ) { ... }
Decompress( ) { ... }
Play( ) { ... }
```



多进程实现方法

- 存在的问题：
 - 进程之间如何通信和共享数据？
 - 系统开销较大：创建进程、进程结束、进程切换

```
程序1
main( )
{
    while(TRUE )
    {
        Read( );
    }
}
Read( ) { ... }
```

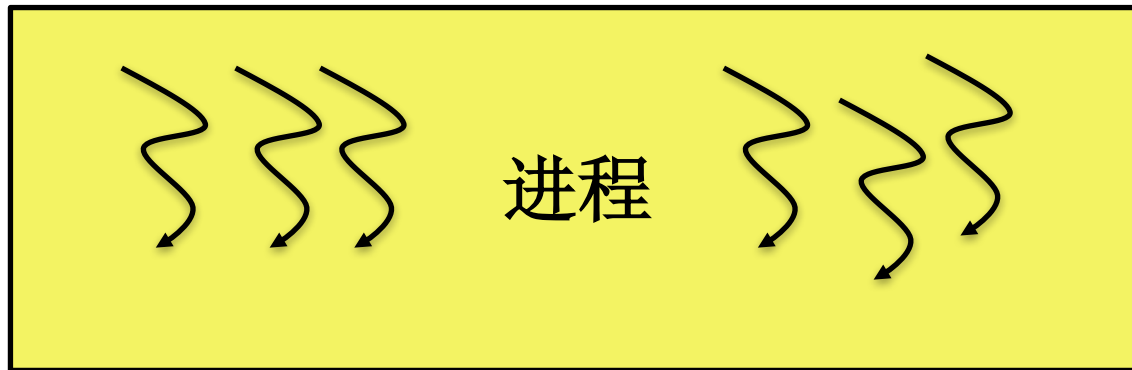
```
程序2
main( )
{
    while(TRUE )
    {
        Decompress( );
    }
}
Decompress( ) { ... }
```

```
程序3
main( )
{
    while(TRUE )
    {
        Play( );
    }
}
Play( ) { ... }
```



引入线程 (Thread)

- 线程是进程的一部分，具有一段执行流
 - 线程1：Read()
 - 线程2：Decompress()
 - 线程3：Play()
- 线程在同一个进程的地址空间内，可共享变量
- 线程是CPU调度的基本单位





IBM System/360引入线程

- **MVT** : Multiprogramming with a Variable number of Tasks





重新审视进程

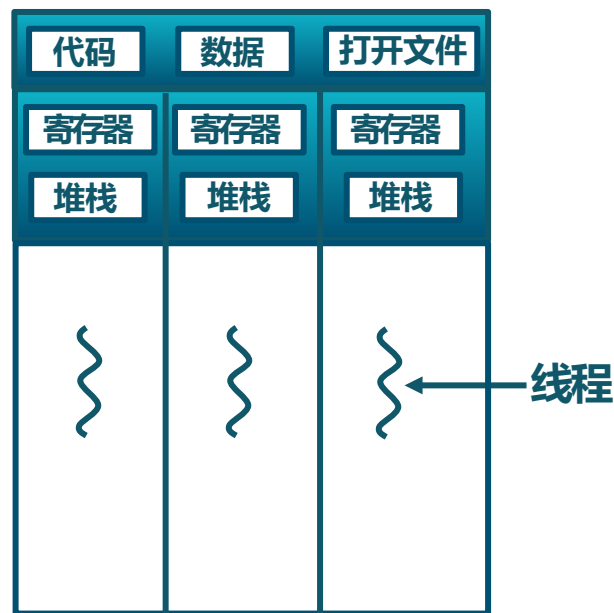
- **进程**

- 运行时 (runtime) : 代码、寄存器、堆、栈
- 资源 : 地址空间 , 文件描述符 , 权限等

- **最简单的进程只有一个线程**



单线程进程



多线程进程



进程 vs. 线程

- **地址空间**

- 进程之间一般不会共享内存
- 进程切换会切换页表和其他内存机制
- 进程中的线程共享整个地址空间

- **权限**

- 进程拥有自己的权限（例如，文件访问权限）
- 进程中的线程共享所有的权限

- **问题**

- 多线程共享进程整个地址空间会有什么问题？



回顾 Intro01 : 示例程序

- 一个程序的运行

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
```

```
int counter = 0;
```

```
int loops;
```

```
void *worker(void *arg) {
    for (int i=0;i<loops;i++) {
        counter++;
    }
    return NULL;
}
```

```
int main(int argc, char *argv[]) {
    loops = atoi(argv[1]);
    printf("Initial value: %d\n", counter);

    pthread_t p1, p2;

    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);

    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    printf("Final value: %d\n", counter);

    return 0;
}
```



过程 vs. 线程

- 过程调用
 - 调用者保存上下文
 - 被调用者保存上下文

save active caller registers

call foo



restore caller regs



```
foo() {  
    do stuff  
}
```

ebp

esp

其他栈帧

调用者保存的寄存器现场

传给过程的参数...

返回地址

调用者EBP

局部变量...

被调用者保存的寄存器现场

高地址

低地址



过程 vs. 线程

- **多线程并发执行**
 - 多线程可以并行地在多个CPU上运行
 - 过程调用是顺序的
- **线程可能会乱序地恢复**
 - 不能用栈（LIFO）恢复线程
 - 每一个线程都有自己的栈
- **线程切换不会太频繁**
 - 不会划分寄存器
 - 线程有“自己”的CPU，通常绑核运行



线程、异步与并发性

- 异步执行
 - 一个操作调用未完成时即可返回
 - 过程调用可以是异步的
 - 多线程可以实现异步执行
- 线程并发性
 - 计算交叠、IO交叠（ I/O overlapping ）：线程可以更容易实现
 - 人们更希望同时做多件事情
 - 例如，服务器（ e.g. 文件服务器, Web服务器，数据库服务器 ）服务多个请求
 - 多个CPU共享内存



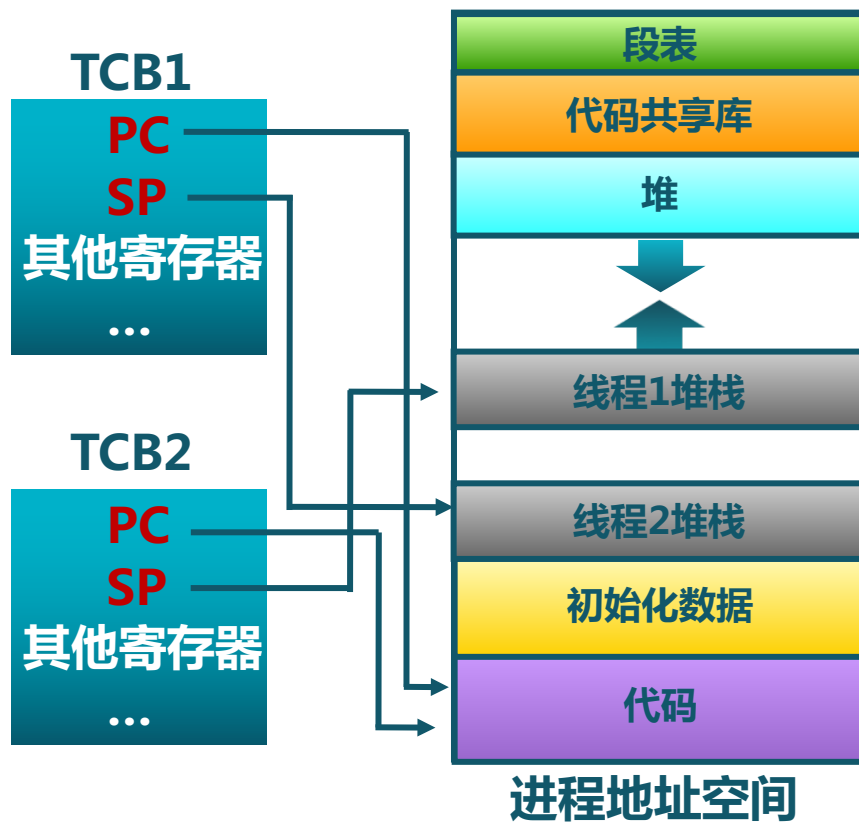
内容提要

- 线程的概念
- 线程表示与操作API
- 线程模型



线程控制块 (TCB)

- 状态
 - 就绪态：准备运行
 - 运行态：正在运行
 - 阻塞态：等待资源
- 寄存器
- 程序计数器(EIP)
- 栈
- 代码





典型的线程API

- **创建**
 - fork, join
- **互斥**
 - acquire(上锁) , release (解锁)
- **条件变量**
 - wait, signal, broadcast
- **警报**
 - alert, alertwait, testalert



线程上下文切换

- **保存上下文**

- 所有的寄存器（通用寄存器和浮点寄存器）
- 所有协同处理器的状态
- 需要保存栈吗？
- Cache和TLB该怎么办？

- **开始新的上下文**

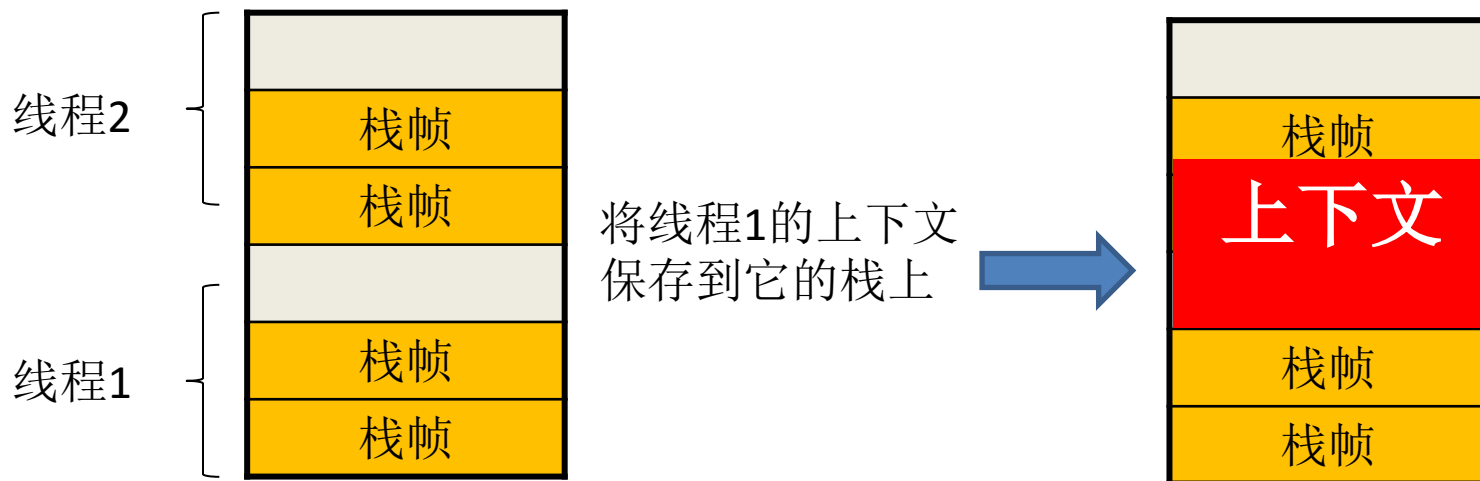
- 相反的操作过程

- **可能触发进程的上下文切换**



保存线程上下文

- **在线程的栈上保存上下文**
 - 许多处理器都有专用的指令来（高效地）保存上下文
 - 但是，需要处理溢出的问题
- **保存前需要检查**
 - 确保栈上没有溢出的问题
 - 把上下文保存到TCB中（residing in the heap）
 - 效率不是很高，但是没有溢出问题





内容提要

- 线程的概念
- 线程表示与操作API
- 线程模型



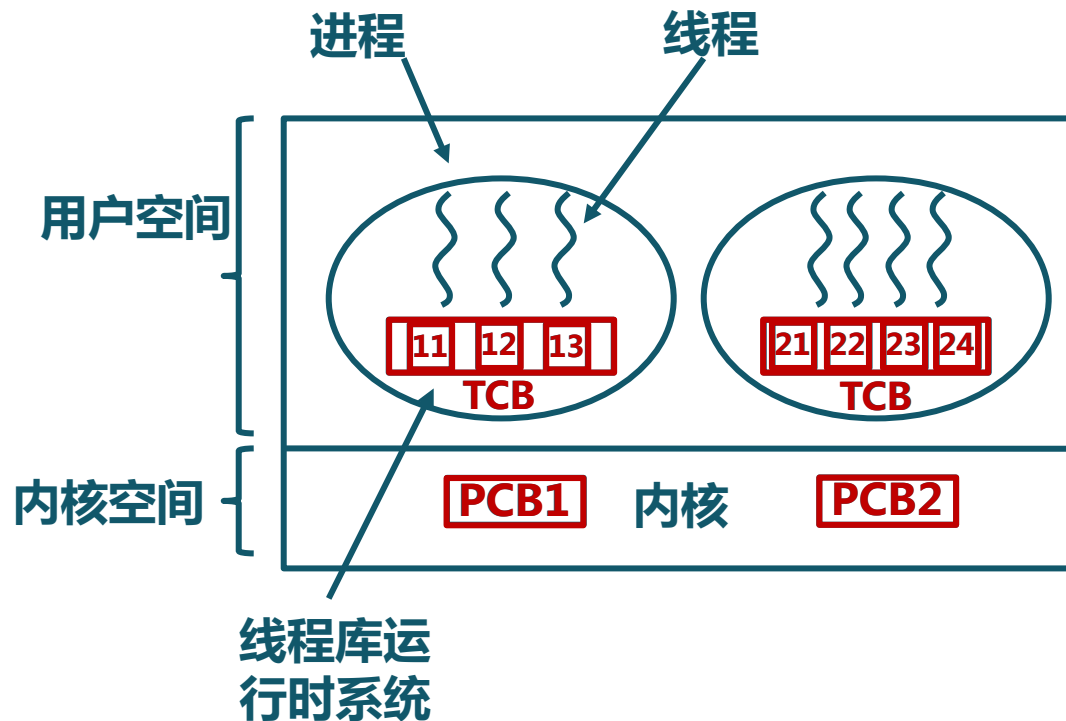
线程模型

- 线程模型
 - 用户级线程 (User-Level Thread)
 - 内核级线程 (核心级线程 , Kernel-Level Thread)
- 分类依据
 - 核外调度 vs. 核内调度
 - 核外调度：减少上下文切换开销
 - 核内调度：充分利用SMP结构



用户级线程

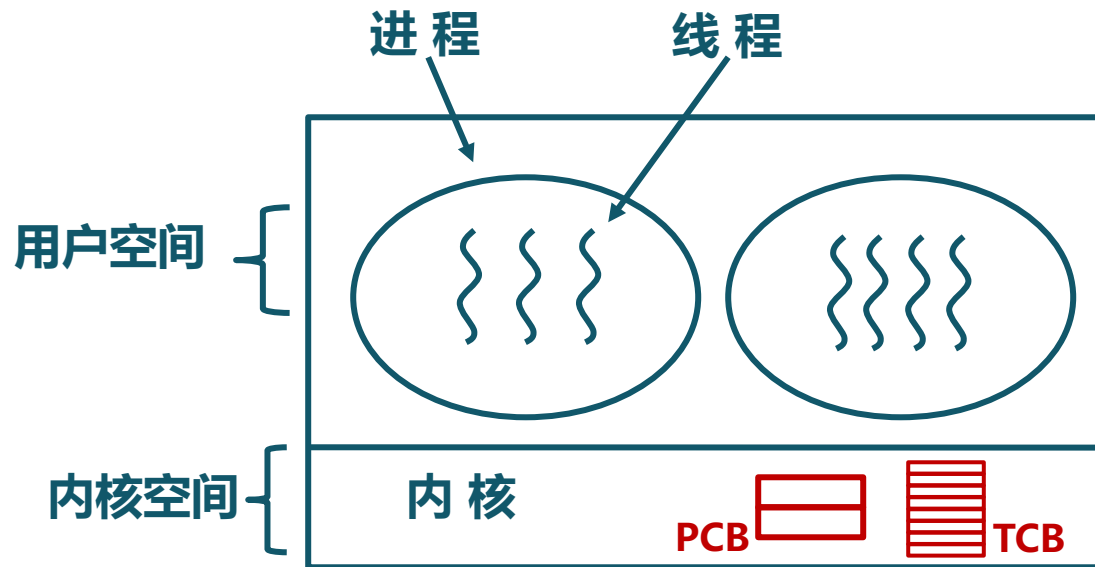
- 由一组用户级的线程库函数来完成线程的管理，包括线程的创建、终止、同步和调度等
- 例如：协程





内核级线程

- 由内核通过系统调用实现的线程机制，由内核完成线程的创建、终止和管理





内核级线程

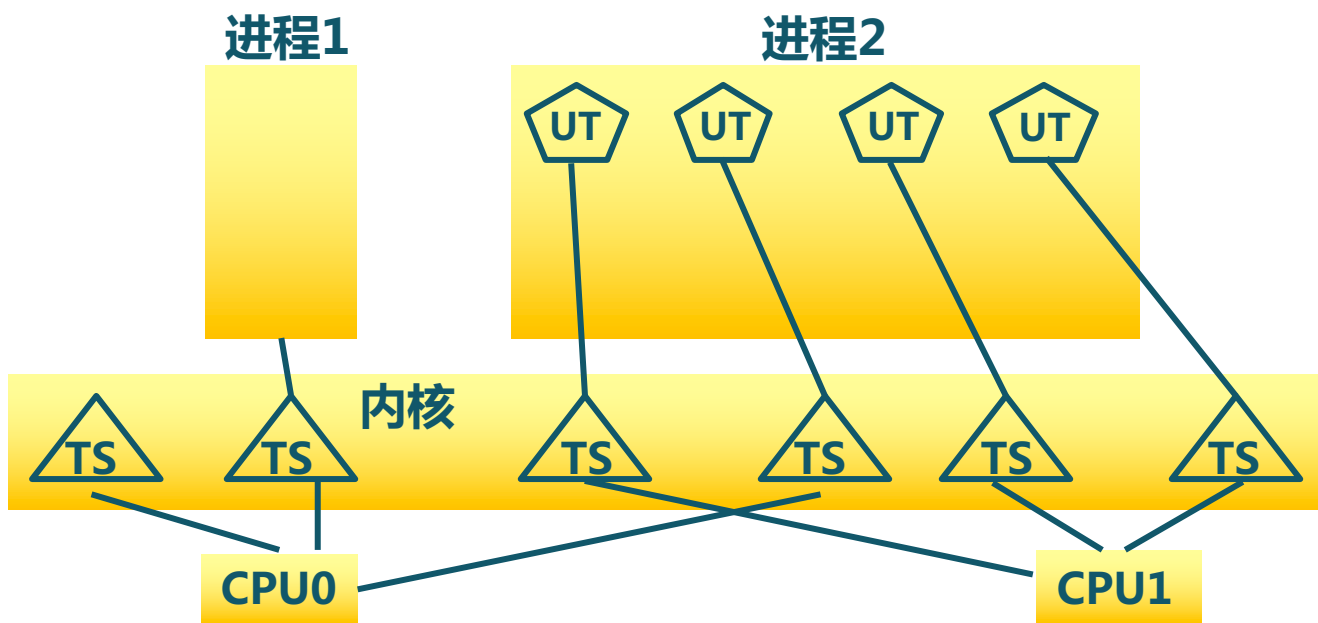
- 内核线程

- 直接由内核本身启动的工作线程（进程），执行内核函数，例如
 - kthreadd: 管理调度其它的内核线程
 - pdflush: 周期性地将修改的内存页写回设备
 - kswapd0: 回收内存页
 - kblockd: 管理系统的块设备，周期性激活系统内的块设备驱动
 - ksoftirqd/n: 处理软中断
- 特点
 - 在CPU特权级运行
 - 访问内存的内核地址空间



轻量级进程 (Light Weight Process, LWP)

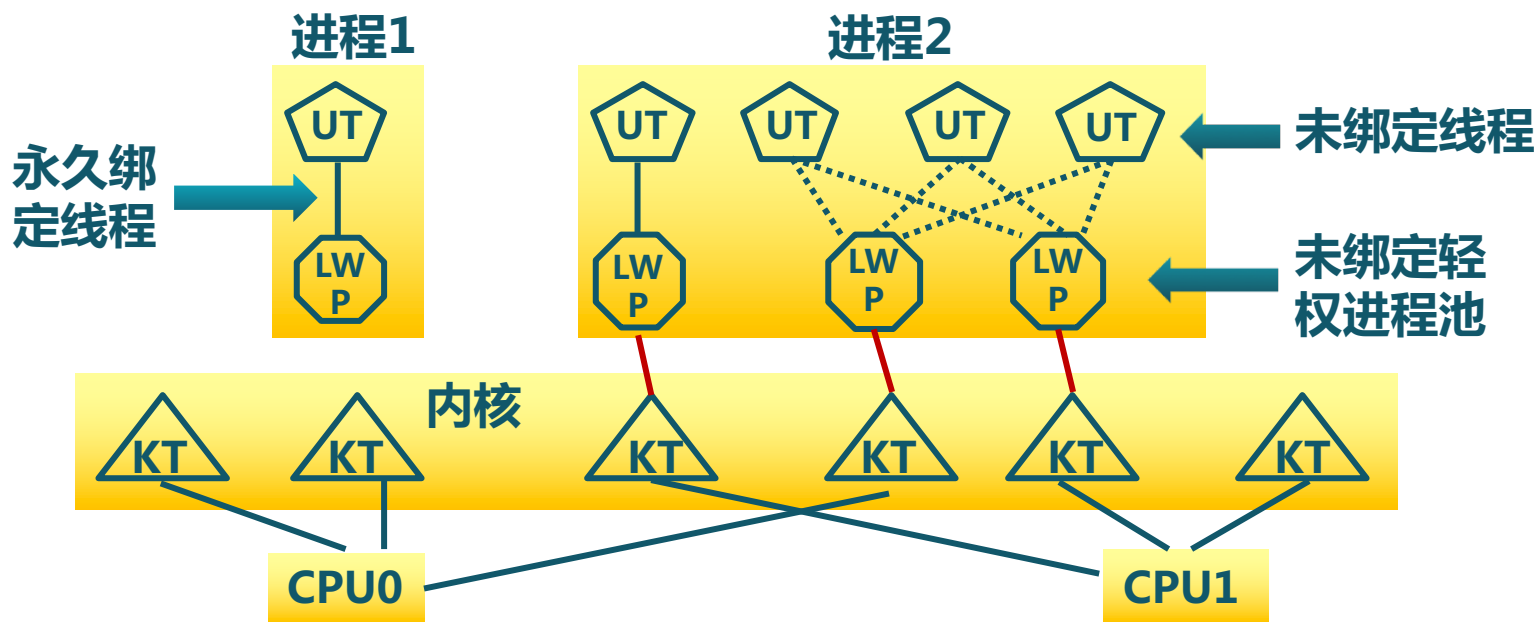
- 模式一：共享某些资源的进程
 - 多个进程共享资源，例如内存、打开文件、信号等。
 - 一个由线程库管理的用户级线程映射到一个LWP上 (Linux)





轻量级进程 (Light Weight Process, LWP)

- 模式二：内核支持的用户级线程
 - 一个进程可有一个或多个轻量级进程，相互共享地址空间等资源，每个轻量级进程由一个单独的内核线程来支持。
 - 多个由线程库管理的用户级线程映射到一个或多个 LWP 上 (Solaris, Unix System V)



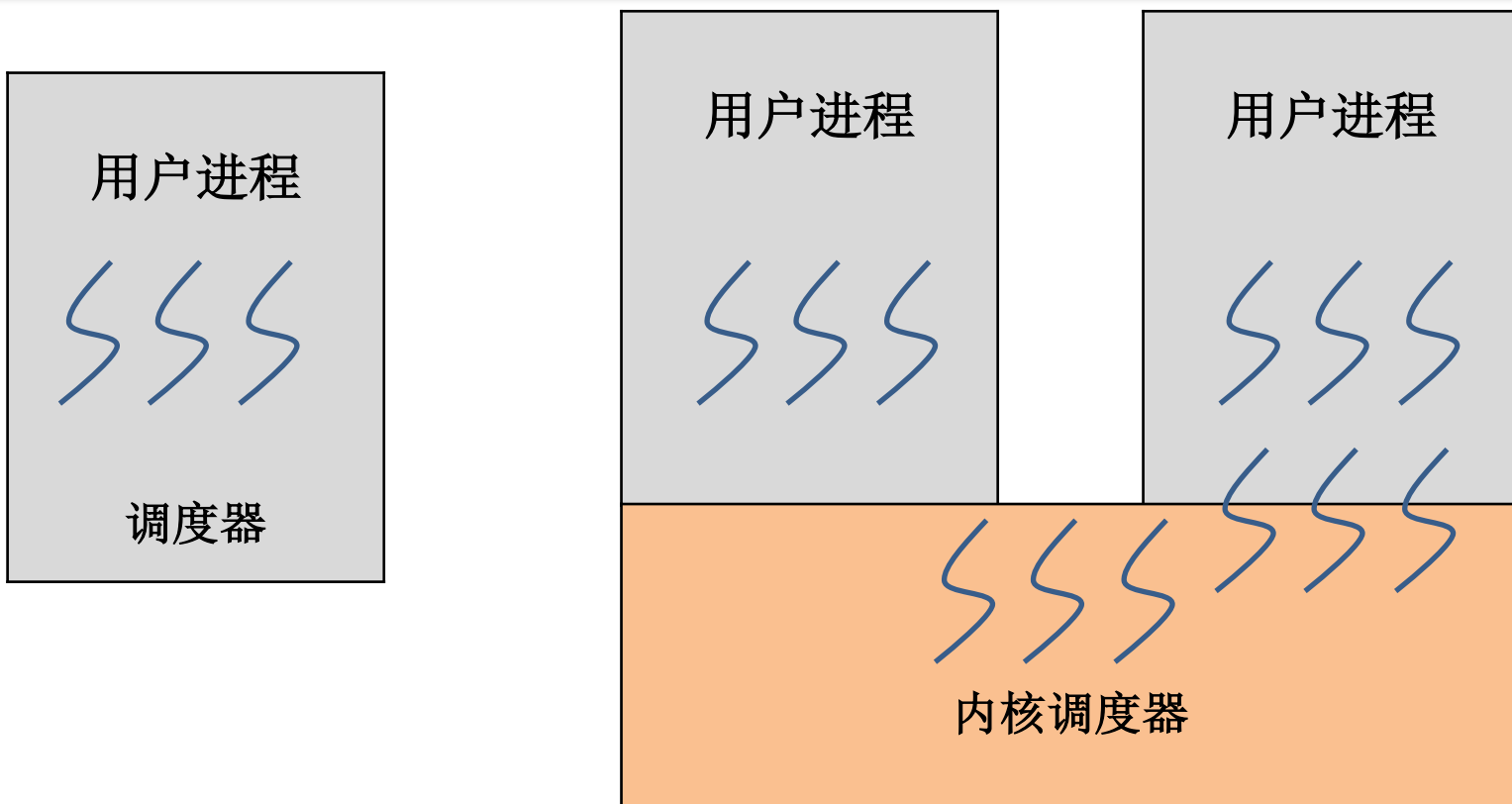


扩展了解：Linux进程/线程

- Linux进程/线程模型
 - task_struct结构体
 - 不区分进程和线程的数据结构
 - 早期：LinuxThreads
 - 专门的管理线程
 - 目前：Native POSIX Thread Library (NPTL)
 - 没有专门的管理线程
 - 某些管理功能由内核直接提供，例如给所有线程发信号



调度：用户级线程 vs. 内核级线程



- 在用户态进行上下文切换，
无需系统调度（python，golang）
- 有可能进行抢占式调度吗？
- 那么I/O事件呢？
- 用户线程
 - 进行系统调用（e.g. I/O）
 - 被中断
- 在内核态进行上下文切换



调度：用户线程 vs 内核线程

- **用户级线程**

- 用户级线程库实现线程上下文切换
- 时间中断会引入抢占
- 当用户级线程被I/O事件阻塞时，整个进程都会被阻塞

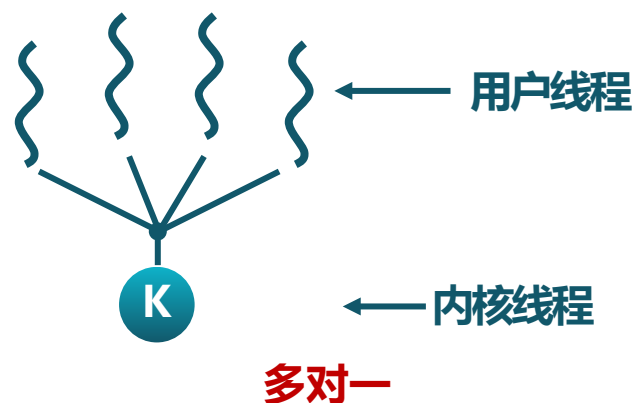
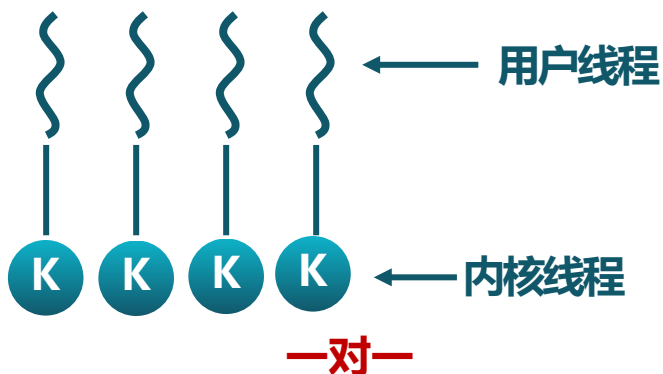
- **内核级线程**

- 内核级线程被内核调度器调度
- 由于跨越了保护边界，内核级线程的上下文切换开销远大于用户级线程



不同线程模型的调度机制

- 主要有两种关系
 - 内核级线程：一对一
 - 用户级线程：多对一
 - 轻量级进程：一对一（Linux），多对多（Solaris）





不同映射关系的对比

- 一对一：每一个线程都拥有自己的内核栈
- 多对一：一个进程的所有线程共享同一个内核栈

	一对一 私有的内核栈	多对一 共享的内核栈
内存利用率	高	低
系统服务	并发存取	串行访问
多处理器	是	无法利用
复杂性	高	低



总结

- 线程的概念
 - 程序调度执行的最小单元
 - 支持应用内部并发性
 - 线程 vs. 进程
 - 线程 vs. 过程
- 线程操作
- 线程模型
 - 内核级线程
 - 用户级线程
 - 轻量级进程