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

操作系统课程 B0911010H-01

线程

中国科学院大学计算机学院

2025-09-29

内容提要

- 线程的概念
- 线程表示与操作API
- 线程模型
- 并发与并行

多进程开发

- 例子1：要写100个文件，每个文件1MB，包括N个数值，可以用多进程来开发实现么？
- 例子2：要读例子1中的100个文件，找出其中的最大值，可以用多进程实现么？
- 例子3：有一个包含100万元素的数组，可以用多进程往其中并行写入数据？

多进程开发

- 存在的问题
  - 进程之间如何共享数据？
  - 系统开销较大：进程创建、结束、切换

```
程序1
main( )
{
   write(file1 )
}
```

```
程序2
main( )
{
   write(file2 )
}
```

…

```
程序100
main( )
{
   write(file100 )
}
```

```
程序1
main( )
{
   read(file1 )
   findMax()
}
```

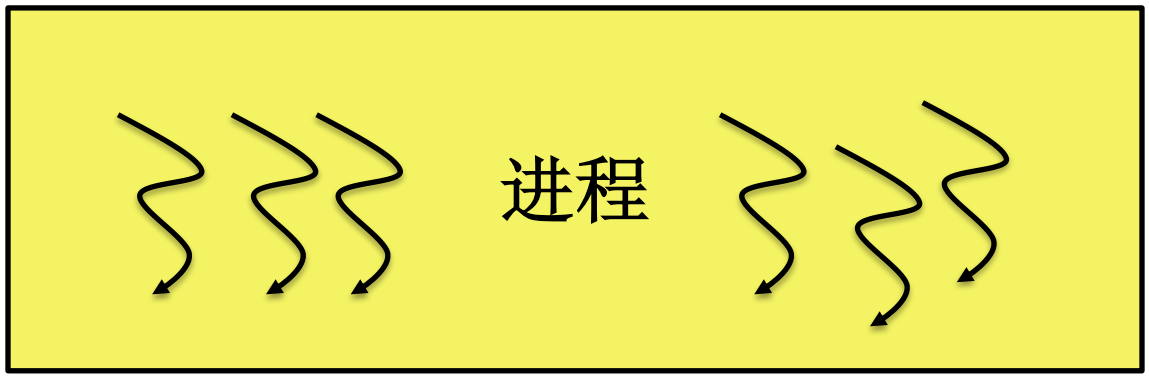
```
程序2
main( )
{
   read(file2 )
   findMax()
}
```

…

```
程序100
main( )
{
   read(file100 )
   findMax()
}
```

引入线程（Thread）

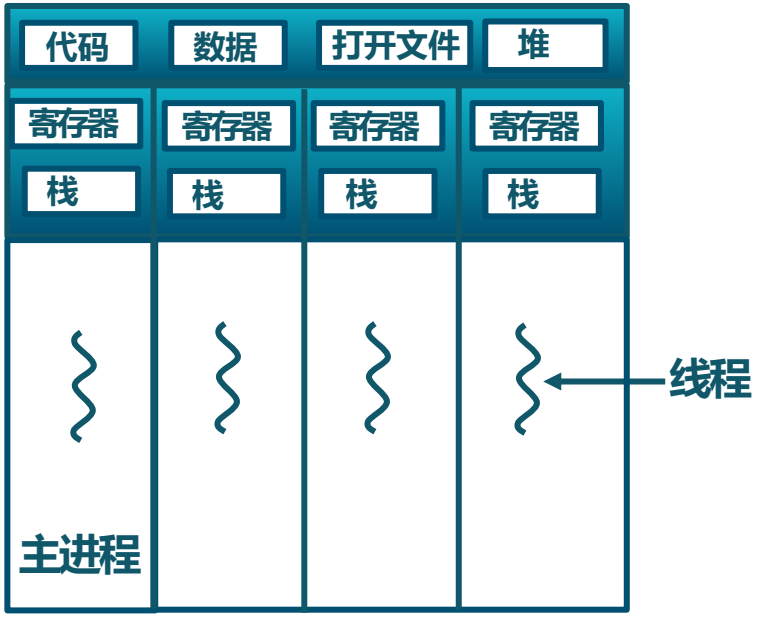
- 线程是进程内部可并行和独立执行的单元（执行流）
  - 线程1：read(file1) → findMax() → max1
  - 线程2：read(file2) → findMax() → max2
  - …
  - 线程100：read(file100) → findMax() → max100
  - 线程101：findMax(max1, max2, …, max100)
- 同一个进程的线程在相同的地址空间内，可共享变量
- 线程是CPU调度的基本单位



重新审视进程

- **进程**
  - 地址空间：代码、数据段、堆、栈
  - 处理器上下文：CPU寄存器
  - 使用资源：内存，文件描述符，权限等
- **最简单的进程只有主进程，可以看成是一个线程**

单线程进程



多线程进程

进程 vs. 线程

- **地址空间**
  - 不同进程之间不共享地址空间
  - 进程中的不同线程共享进程的整个地址空间
- **资源**
  - 进程描述了程序运行过程中使用的资源（例如，内存地址空间、打开文件、占用网络端口、访问权限等）
  - 进程中的线程共享使用进程的资源
- **问题**
  - 多线程共享进程整个地址空间会有什么问题？

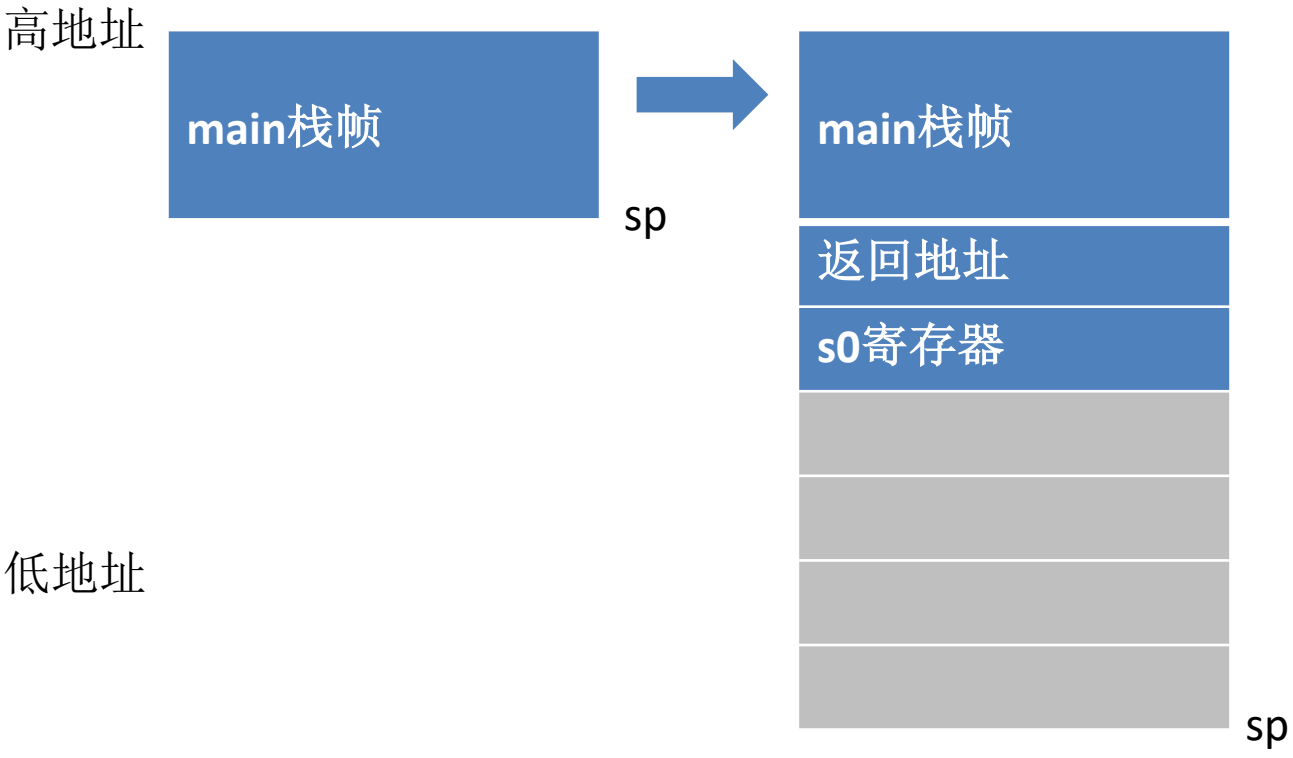
过程 vs. 线程

- 过程调用
  - 创建一个新的栈帧
  - 被调用者将返回地址压栈、s0寄存器压栈

以RISC-V 32 + clang为例

```
foo()  {
   do  stuff
}

main() {
   foo()
}
```



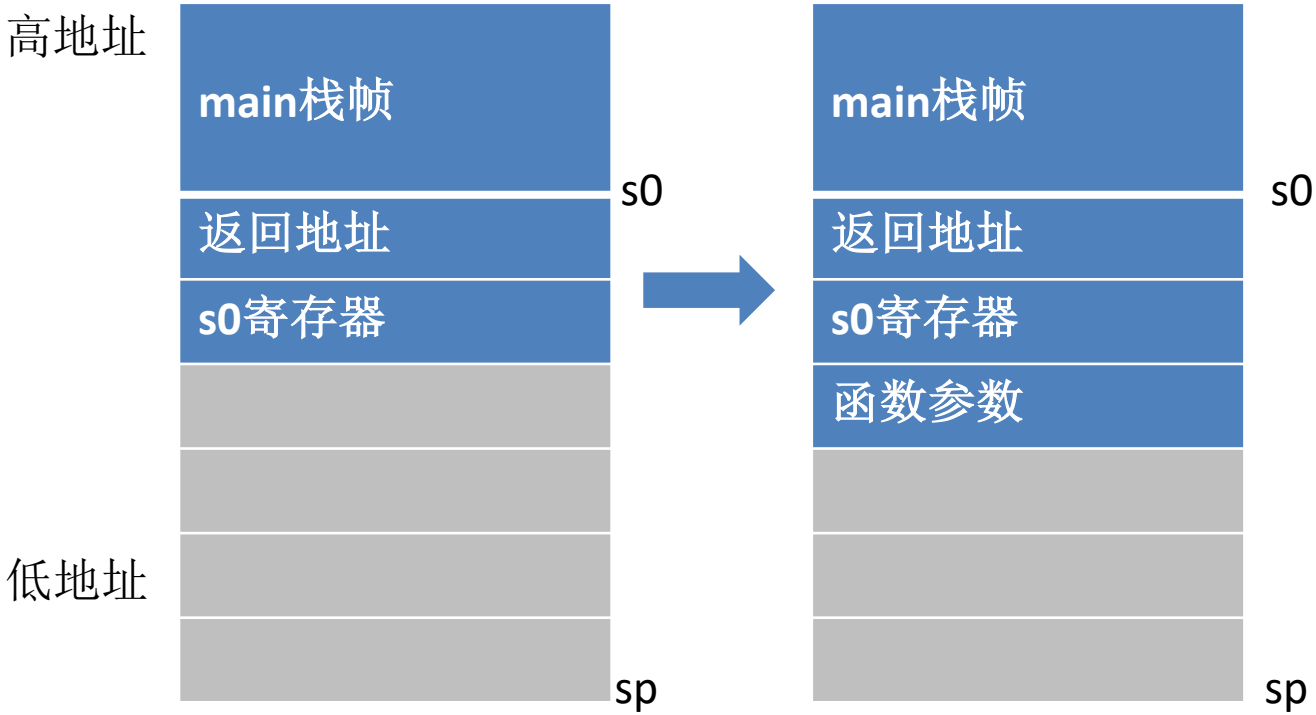
过程 vs. 线程

- 过程调用
  - s0指向栈底，sp指向最新的栈顶
  - 被调用者将函数参数压栈

以RISC-V 32 + clang为例

```
foo()  {
   do  stuff
}

main() {
   foo()
}
```



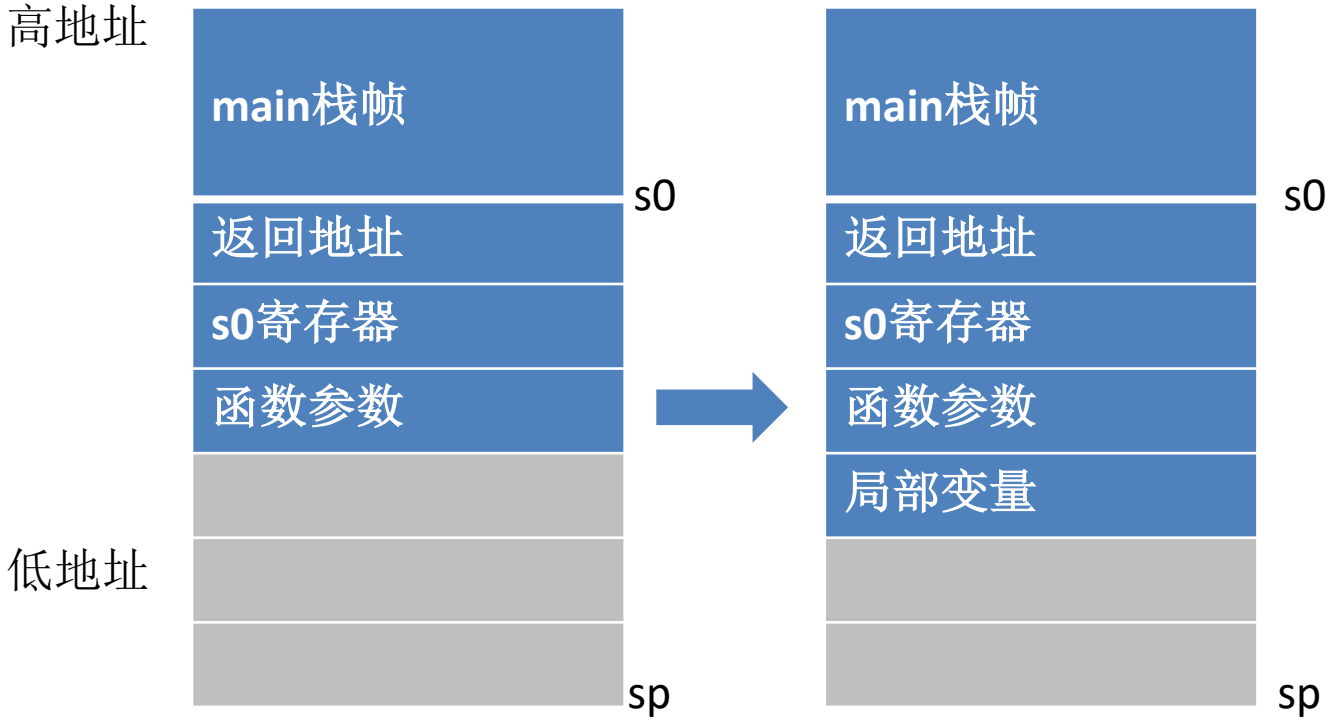
过程 vs. 线程

- 过程调用
  - 被调用者将局部变量压栈

以RISC-V 32 + clang为例

```
foo()  {
   do  stuff
}

main() {
   foo()
}
```



过程 vs. 线程

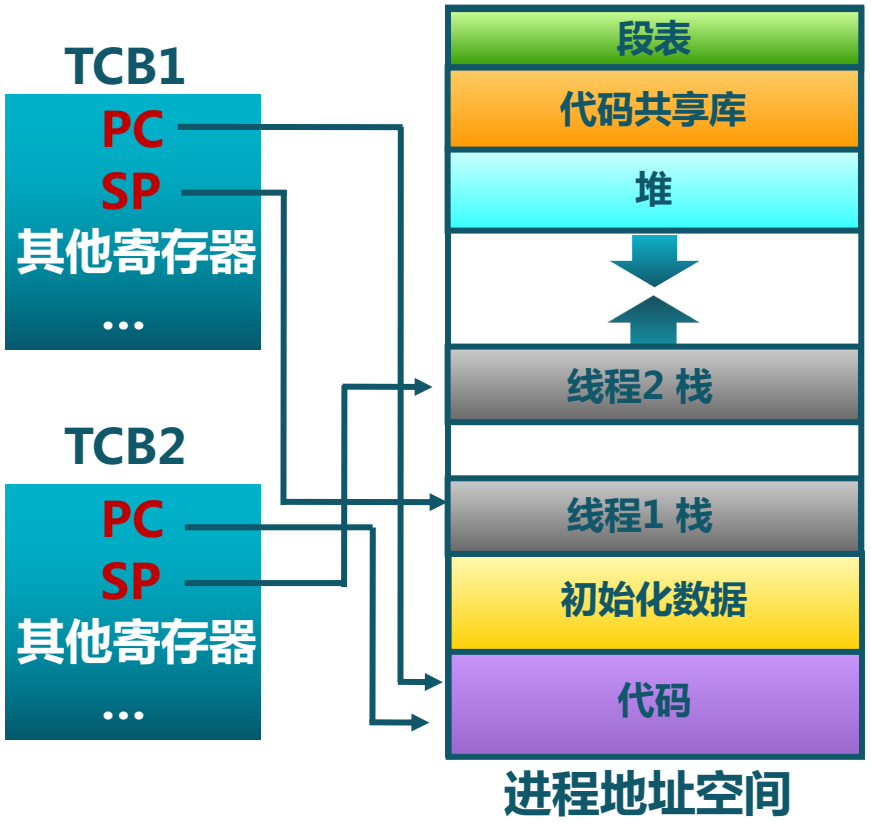
- 多线程可以并行执行
  - 多线程可以并行地在多个CPU核上运行
  - 过程调用是顺序的
- 线程是独立的调度实体，过程不是
  - 每一个线程都有自己的运行时上下文（PC、SP寄存器）、栈，可以被独立调度
  - 调度器按调度算法调度线程，导致线程执行顺序具有不确定性
  - 线程可以和CPU绑定运行，即绑核
    - 扩展了解：cpu_set_t类型和sched_setaffinity函数

内容提要

- 线程的概念
- 线程表示与操作API
- 线程模型
- 并发与并行

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
  - PC, SP寄存器
- 栈
- 代码



典型的线程API

- 以Linux的pthread库为例
- 创建
  - pthread_create: 通过clone系统调用创建线程
    - int clone(int (*fn)(void *), void *stack, int flags, void *arg, ...)
    - CLONE_VM：共享内存地址空间
    - CLONE_FILES：共享文件描述符表
  - pthread_join：等待线程结束，阻塞调用该函数的线程，直到指定线程结束执行
- 互斥
  - acquire(上锁)，release（解锁）
- 条件变量
  - wait, signal, broadcast

线程上下文切换

- 保存上下文
  - 保存通用寄存器（PC、SP寄存器）
  - 保存协处理器的状态
  - 需要刷CPU Cache么？
  - 地址空间（页表）需要切换么？
- 开始新的上下文
  - 相反的操作过程

内容提要

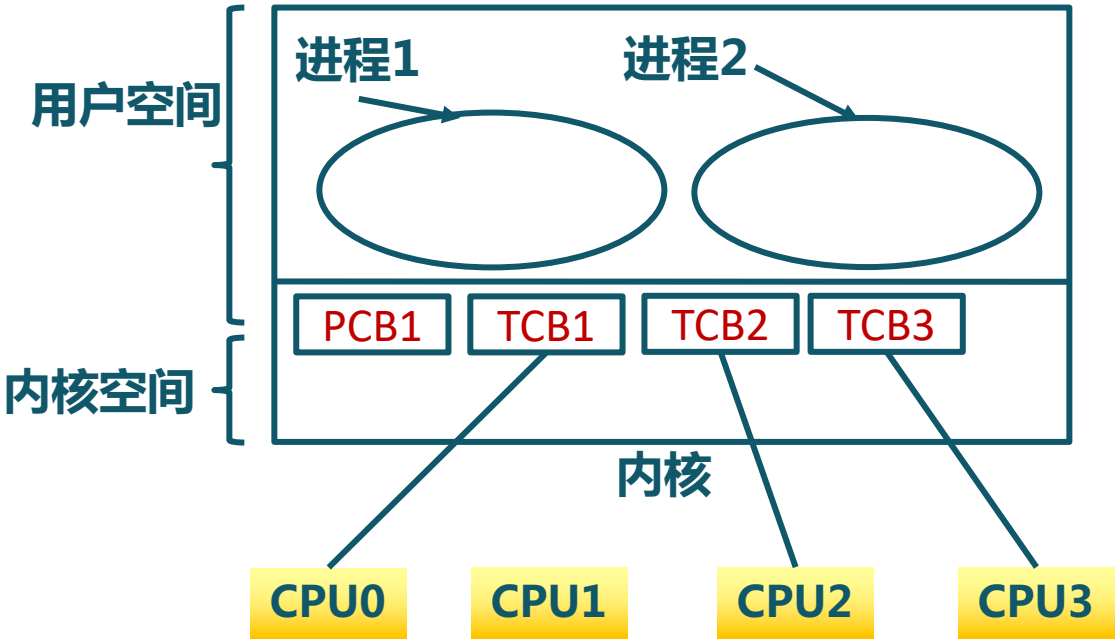
- 线程的概念
- 线程表示与操作API
- 线程模型
- 并发与并行

线程模型

- 线程模型
  - 内核级线程（Kernel-Level Thread）
    - 内核负责线程的创建、调度和管理
  - 用户级线程（User-Level Thread）
    - 线程库负责线程的创建、调度和管理
  - 轻量级进程（Light-weight Process，LWP）

内核级线程

- 直接由内核创建、调度和管理的线程
  - 每个线程对应一个内核调度实体，可以在不同CPU核上并行执行
  - 内核为每个线程维护一个管理结构（TCB）

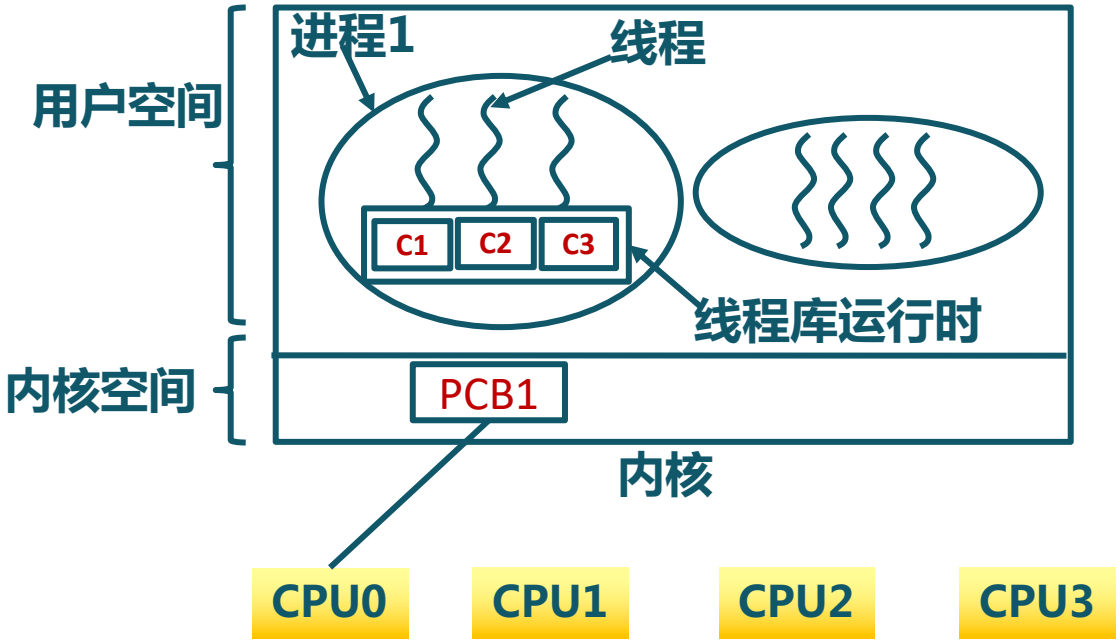


内核线程

- 内核本身启动的工作任务，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
  - 访问内核地址空间

用户级线程

- 内核级线程的管理涉及内核，创建和切换开销大
- 由用户级线程库创建、调度、切换和管理的线程为用户级线程
  - 协程可以看成是一种形式的用户级线程
  - 通常结合异步IO实现高并发任务执行

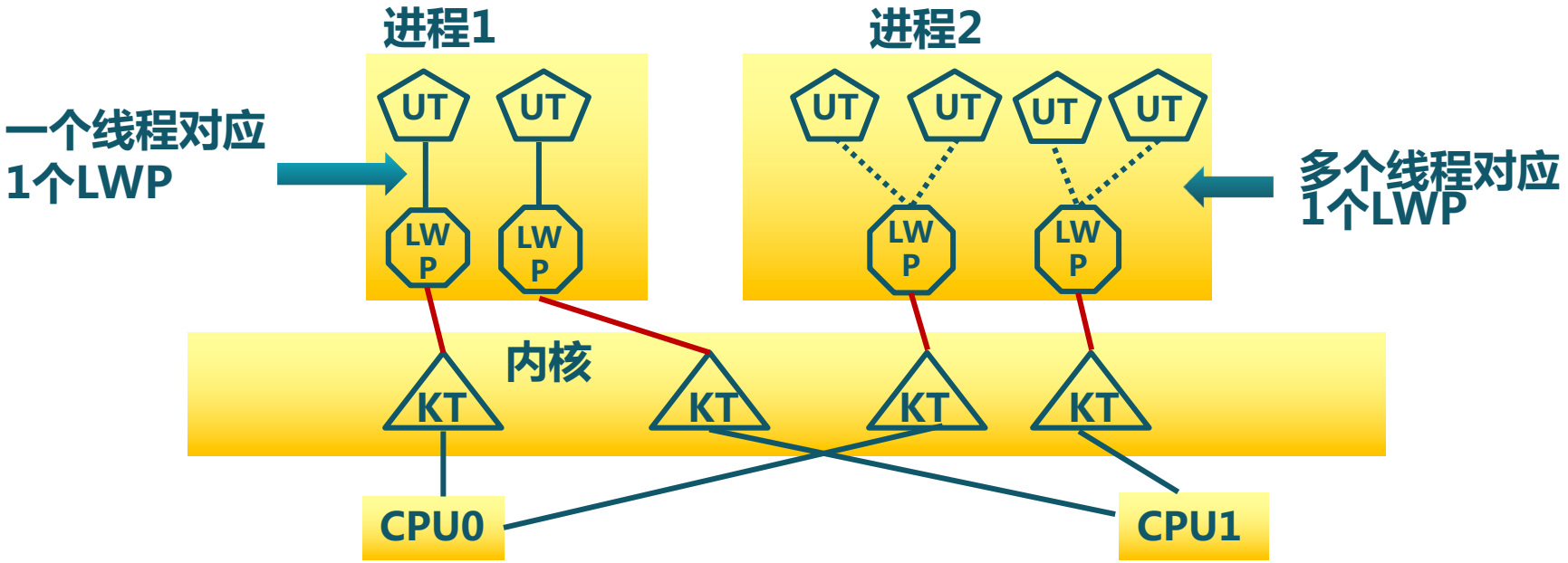


轻量级进程（Light-weight Process, LWP）

- 内核级线程不足
  - 线程的管理都要经过内核
- 用户级线程不足
  - 对内核不可见，无法充分利用多核并行性
  - 一旦某个用户级线程被阻塞，其他线程都被阻塞
- 轻量级进程混合使用内核级线程和用户级线程，获得两者优势，避免两者不足
  - 由父进程创建，可以和父进程共享某些资源，例如地址空间、打开文件等资源
  - 每一个LWP对内核可见、可调度

轻量级进程（Light-weight Process, LWP）

- 用户级线程与LWP映射模式
  - 用户级线程按不同模式映射到LWP
  - 模式一：一个用户级线程映射到一个LWP (Linux)
  - 模式二：多个用户级线程映射到一个LWP（Solaris）

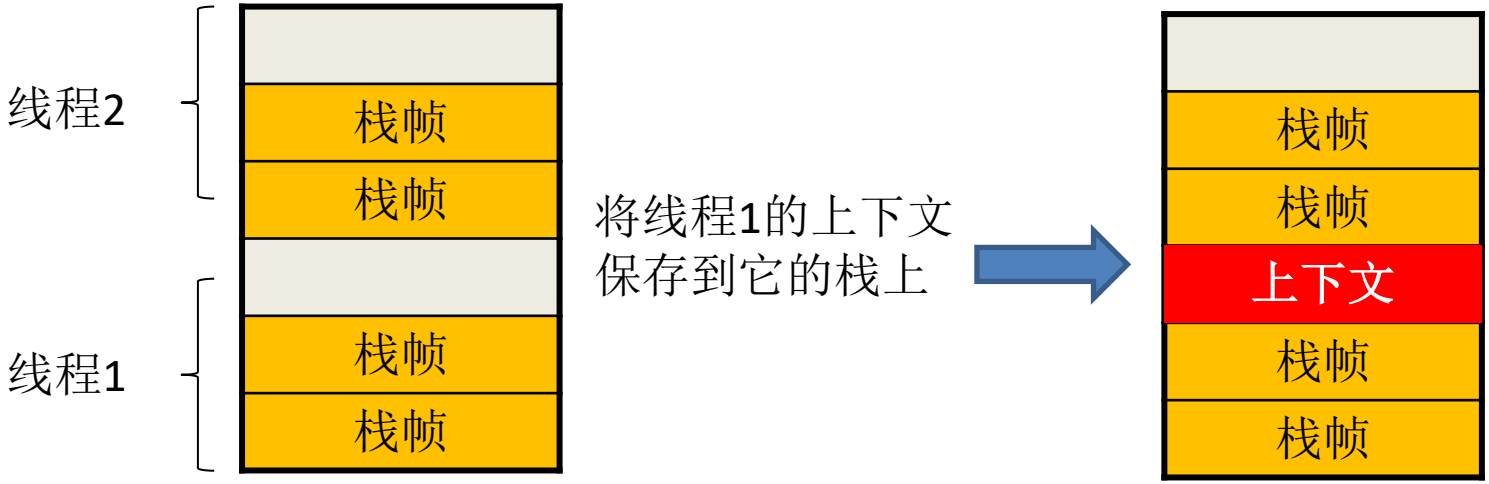


扩展了解

- Linux进程/线程模型
  - PCB/TCB统一用task_struct结构体
  - 不区分进程和线程的数据结构
- pthread库
  - 早期
    - 由库实现用户级线程调度执行
  - 发展
    - 在用户态创建用户级TCB
    - 使用clone系统调用直接在内核创建内核级线程，创建内核级TCB（task_struct）
    - 底层依赖Native POSIX Thread Library (NPTL)实现

使用LWP时如何保存线程上下文

- 在线程的内核栈上保存上下文
  - PC、SP、状态寄存器、通用寄存器等
- 内核栈指针保存到内核的TCB上，便于恢复时定位



内核级线程 vs. 用户级线程 vs LWP

- 内核级线程
  - 在内核态实现上下文切换
  - 内核调度器负责调度
  - 单个线程被阻塞不影响其他线程
- 用户级线程
  - 在用户态实现上下文切换
  - 用户级线程库负责调度
  - 某个线程被阻塞时（I/O事件），其他线程都会被阻塞
- LWP
  - 在内核态实现上下文切换
  - 内核调度器负责调度
  - 映射的用户级线程阻塞时，只影响同一个映射组内的其他线程

内核级线程 vs. 用户级线程 vs LWP

- 内核级线程：每一个线程对应一个内核调度实体
- 用户级线程：一个进程的所有线程共享一个内核调度实体
- LWP：一个或多个线程对应一个内核调度实体

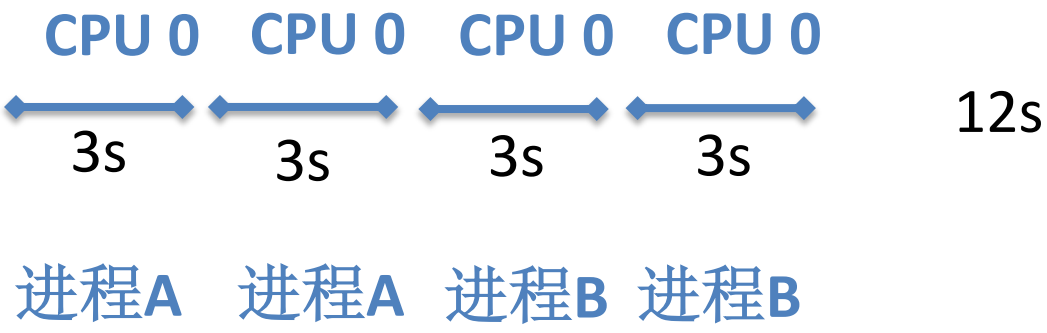
	内核级线程	LWP	用户级线程
内核内存消耗	高	中	低
系统服务	并发	部分并发	串行访问
多处理器	是	部分利用	无法利用
内核复杂性	高	高	低

内容提要

- 线程的概念
- 线程表示与操作API
- 线程模型
- 并发与并行

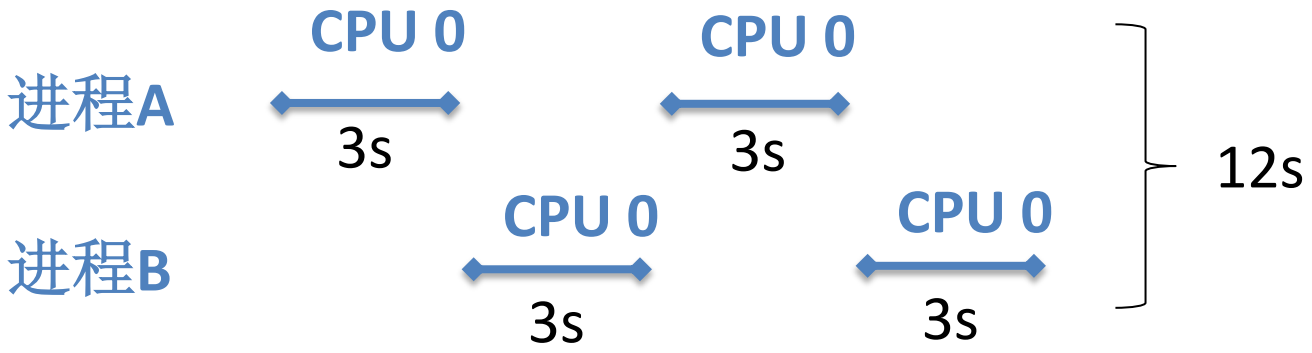
进程的顺序执行

- 示例
  - 进程A和进程B都需要6s完成任务执行
  - 时间片是3s，进程A和B依次执行，没有并发和并行



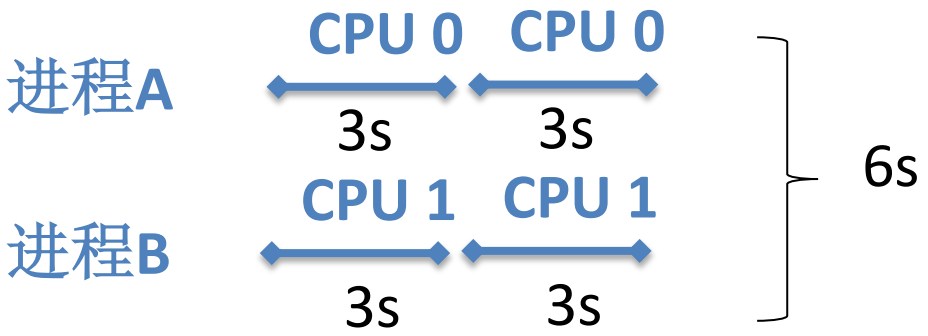
进程和并发性

- 并发性
  - 一个系统中有多个进程“同时”运行（逻辑上）
  - CPU、DRAM和I/O设备是共享的
  - 每一个进程都希望能拥有自己的计算机资源
- 虚拟化（分时复用）
  - 每个进程都运行一段时间（时间片）
  - 使得一个CPU变成“多个”，每一个进程就好像拥有了自己的CPU

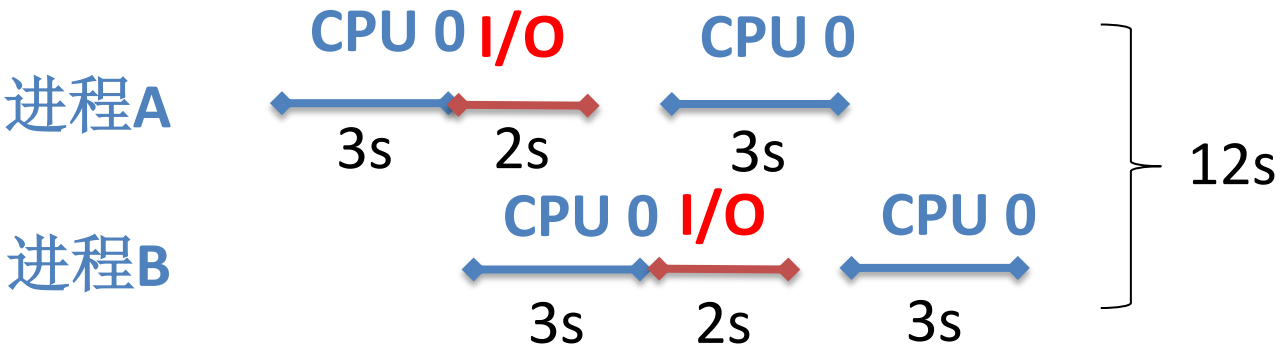


进程和并行性

- 并行性
  - 多个CPU
  - 进程在物理上同时运行
  - 用途：加速



- CPU和I/O的并行
  - CPU计算与I/O操作交叠
  - 减少总共所需完成时间
  - 如果没有CPU和I/O并行，需要多长时间？



并发和并行

- 并发
  - 一个系统能同时处理多个任务的能力，但同一时刻可能只有一个任务在运行
- 并行
  - 一个系统在同一时刻支持多个任务同时运行
- 并行处理收益
  - 将一个复杂问题分解成多个子问题
  - 每个子问题由一个进程处理
  - 多个进程同时处理，减少处理时间

线程与并发性/并行性

- 线程可以提升应用的并发性/并行性
  - 线程可以被独立调度用于执行单独的任务
    - 在同一CPU上被调度
    - 被调度到不同CPU上
  - 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
  - 可以实现计算交叠、IO交叠
    - 不同线程在使用不同的资源
  - 多个线程共享内存，并发/并行访问会带来共享资源访问问题

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;
}
```

并行性收益

- 并行性在日常生活中很普遍
  - 1个销售员一年的销售额是10万
  - 雇佣10个销售员就可以售出100万
- 加速比
  - 理想情况下可以获得N倍加速比
  - 现实：各种瓶颈 + 协调开销
- 问题
  - 你和1个伙伴合作可以加速完成任务吗？
  - 你和20个伙伴合作可以加速完成任务吗？
  - 你可以获得超线性的加速比吗（大于N倍）？

总结

- 线程的概念
  - 内核调度执行的最小单元，支持多任务并行执行
  - 线程共享进程的地址空间和资源
- 线程操作
  - pthread线程库
- 线程模型
  - 内核级线程
  - 用户级线程
  - 轻量级进程
- 并发和并行
  - 并发：逻辑上同时运行
  - 并行：物理上同时运行，需要有多资源支持