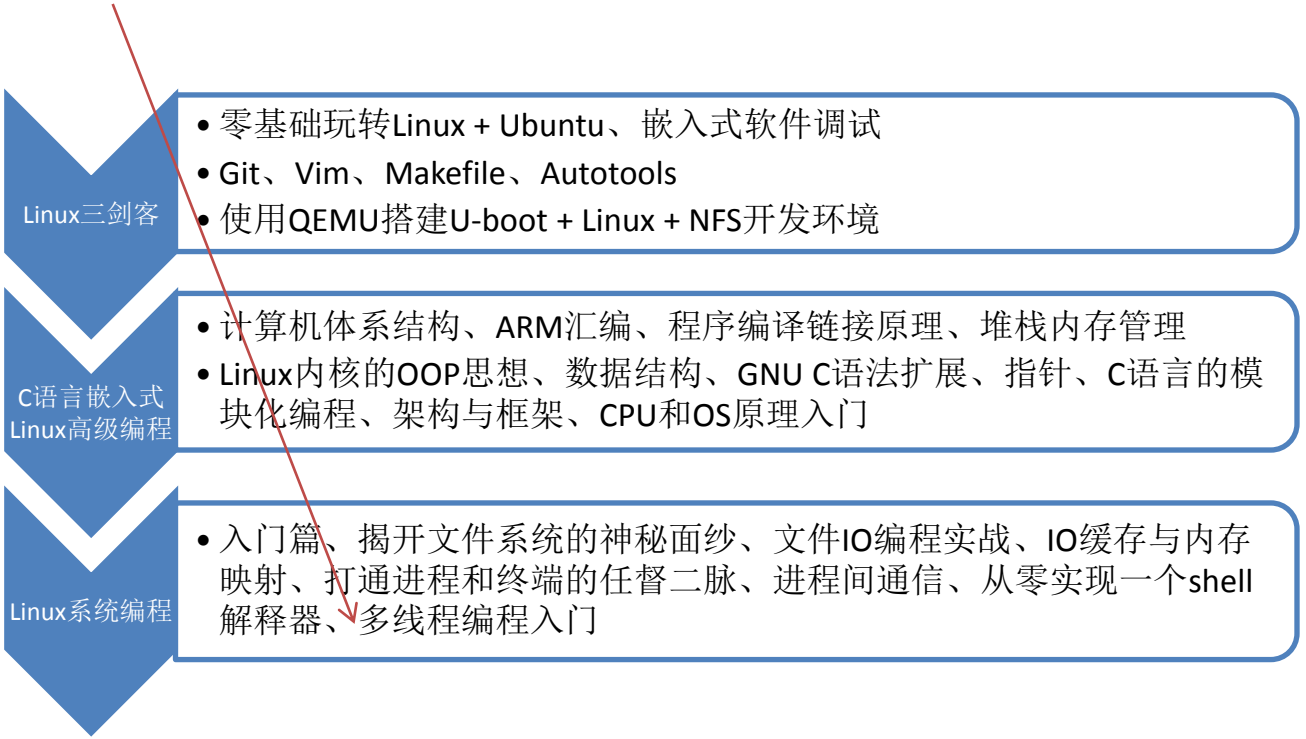
文档版本	说明	作者	创建日期
V0.1	Linux系统编程：入门篇视频配套PPT	王利涛	2018年10月14日
V0.2	第01期：揭开文件系统的神秘面纱	王利涛	2018年11月07日
V0.3	第02期：文件IO编程实战	王利涛	2018年11月25日
V0.4	第03期：IO缓存与内存映射	王利涛	2018年12月11日
V0.5	第04期：打通进程和终端的任督二脉	王利涛	2019年03月15日
V0.6	第05期：进程间通信	王利涛	2019年06月15日
V0.7	第06期：从零实现一个shell解释器	王利涛	2019年08月20日
V0.8	第07期：多线程编程入门	王利涛	2019年12月20日

《嵌入式工程师自修养》视频教程

- 第00步：Linux三剑客
- 第01步：C语言嵌入式Linux高级编程
- 第03步：Linux系统编程
- 第04步：Linux内核编程
- 第05步：嵌入式驱动开发
- 第06步：项目实战
- ----------
- 详情咨询QQ：3284757626
- 视频淘宝店：https://wanglitao.taobao.com
- 博客：www.zhaixue.cc
- 微信公众号：

嵌入式学习路线图

- We are here…

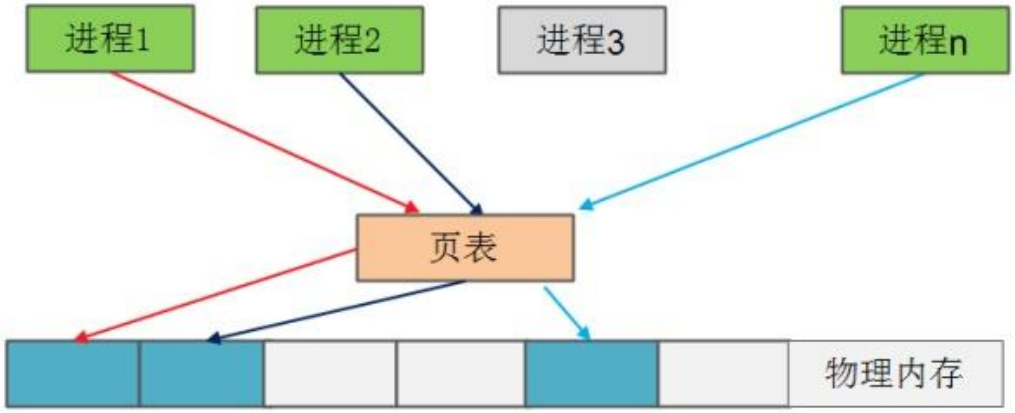


Linux系统编程第07期

多线程编程入门

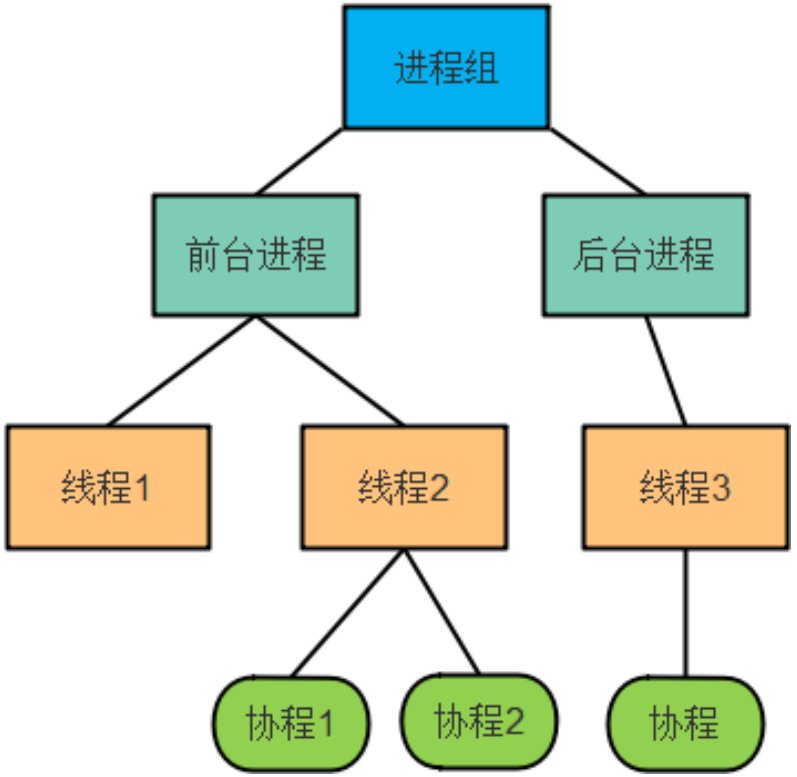
多线程编程的概念

- 关于多线程…
  - 有了进程，为什么还要多线程？
  - 多线程编程有哪些优点？
  - 多线程编程主要用在什么地方？
  - SMP、NUMA、MPP
  - 多核、4核8线程



多线程编程概念

- 进程、线程、协程
  - 提高程序运行效率
  - 模块细分，防止程序阻塞
  - 高并发、多核、服务器
  - 线程池、协程



多线程学习准备工作

准备工作

- Pthread线程库
  - 线程的实现: windows、Linux
  - Pthread库: POSIX标准中的thread API
  - Glibc 与 LinuxThread
  - Glibc 与 NPTL
  - $ getconf GNU_LIBPTHREAD_VERSION

Linux与Windows的API

对象	操作	Linux Pthread API	Windows API
线程	创建	pthread_create	CreateThread
	退出	pthread_exit	ThreadExit
	等待	pthread_join	WaitForSingleObject
互斥锁	创建	pthread_mutex_init	CreateMutex
	销毁	pthread_mutex_destroy	CloseHandle
	加锁	pthread_mutex_lock	WaitForSingleObject
	解锁	pthread_mutex_unlock	ReleaseMutex
条件变量	创建	pthread_cond_init	CreateEvent
	销毁	pthread_cond_destroy	CloseHandle
	触发	pthread_cond_signal	SetEvent
	广播	pthread_cond_broadcast	SetEvent / ResetEvent
	等待	pthread_cond_wait / pthread_cond_timedwait	SingleObjectAndWait
读写锁	创建	pthread_rwlock_init	InitializeSRWLock
	等待	Pthread_rwlock_rdlock	AcquireSRWLockShared
	销毁	Pthread_rwlock_destroy	ReleaseSRWLockShared

准备工作

- 使用pthread库
  - 安装man手册
    - $ apt install glibc-doc manpages-posix-dev
  - 程序的编译
    - $ gcc main.c -lpthread
    - /usr/lib/libpthread.a
  - pthread常用API
    - pthread_create
    - pthread_exit
    - pthread_cancel
    - pthread_join
    - pthread_detach
    - pthread_mutex_lock
    - pthread_cond_init
    - pthread_cond_signal
    - pthread_cond_wait
    - pthread_rwlock_rdlock

创建一个线程：pthread_create

创建一个线程

- API接口说明
  - 函数原型：int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);
  - 函数功能：创建一个新线程
  - 参数说明：
    - thread：指向线程ID的指针
    - attr：　线程的属性
    - start_routine：线程执行实体入口
    - arg：　传递给线程的参数
    - typedef unsigned long int pthread_t

线程的终止

线程的终止

- 终止线程的三种方式
  - 从start_routine正常return
  - 显式调用 pthread_exit
    - 函数原型： void pthread_exit(void *retval);
    - 返回值通过参数retval传递
  - 线程被 pthread_cancel取消

TIPS

- 线程pthread_exit与exit的区别
  - 线程调用pthread_exit，只会结束当前线程，不影响进程的运行
  - 一个进程中的任何一个线程调用exit，将会结束整个进程

等待线程的终止

等待线程的终止

- 线程分两种
  - Joinable
    - PTHREAD_CREATE_JOINABLE
    - 可通过 pthread_join 等待线程终止
    - 调用pthread_join 的线程会阻塞
    - 一个Joinable线程结束时，资源不会自动释放给系统(堆栈、exit状态等)
    - 当线程终止时， pthread_join 会回收该线程的资源，然后返回
    - 若无pthread_join参与“擦屁股”工作，该线程将变为僵尸线程
  - Unjoinable
    - PTHREAD_CREATE_DETACHED
    - 可通过 pthread_detach 分离一个线程
    - 当线程终止时，资源会自动释放给系统

等待线程的终止

- API接口
  - 函数原型：int pthread_join (pthread_t thread, void **retval);
  - 函数功能：阻塞掉当前线程，等待指定线程终止
  - 函数终止： void pthread_exit (void *retval);
  - 参数：
    - thread：线程的ID
    - retval：从pthread_exit 返回的值

等待线程的终止

- API接口
  - 函数原型： int pthread_detach(pthread_t thread);
  - 函数功能：将指定线程与当前线程分离
  - 参数说明：指定要分离的线程的ID

线程属性

线程属性

- 默认属性
  - 调度参数：
  - 线程栈地址：
  - 线程栈大小：8M
  - 栈末尾警戒缓冲区大小：PAGESIZE
  - 线程的分离状态：joinable、detached
  - 继承性：PTHREAD_INHERIT_SCHED、PTHREAD_EXPLICIT_SCHED
  - 作用域：PTHREAD_SCOPE_PROCESS、PTHREAD_SCOPE_SYSTEM
  - 调度策略：SCHED_FIFO、SCHED_RR、SCHED_OTHER

```
struct __pthread_attr
{
  struct sched_param __schedparam;
  void *__stackaddr;
  size_t __stacksize;
  size_t __guardsize;
  enum __pthread_detachstate __detachstate;
  enum __pthread_inheritsched __inheritsched;
  enum __pthread_contentionscope __contentionscope;
  int __schedpolicy;
};
```

线程属性

- 相关API函数
  - int pthread_attr_init (pthread_attr_t *attr);
  - int pthread_attr_destroy (pthread_attr_t *attr);
  - int pthread_attr_setstacksize (pthread_attr_t *attr, size_t stacksize);
  - int pthread_attr_getstacksize (const pthread_attr_t *attr, size_t *stacksize);
  - int pthread_attr_setstackaddr (pthread_attr_t *attr, void *stackaddr);
  - int pthread_attr_getstackaddr (const pthread_attr_t *attr, void **stackaddr);
  - int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate);
  - int pthread_attr_getdetachstate (const pthread_attr_t *attr, int *detachstate);

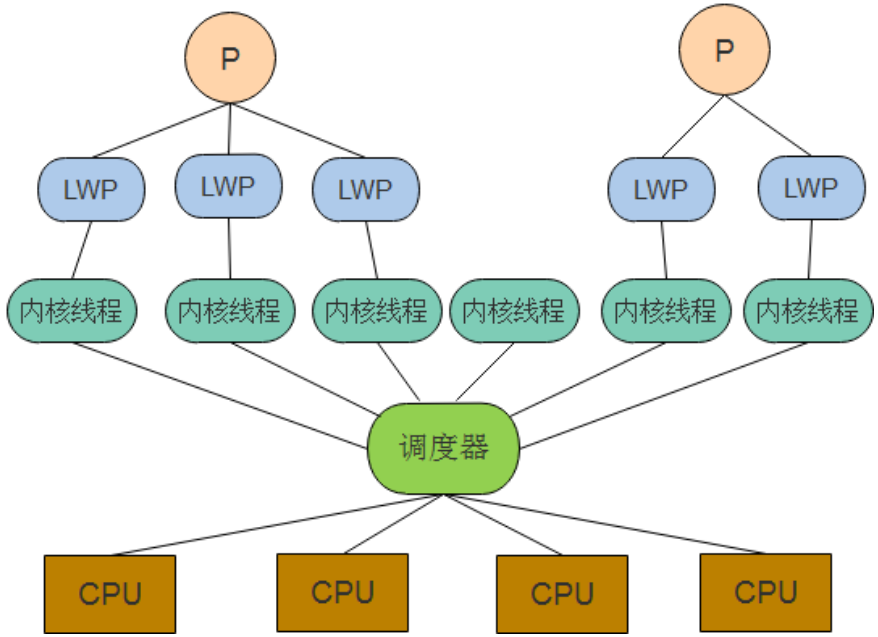
线程调度与运行

线程的调度与运行

- 线程分类
  - 核心级线程
    - 由内核调度，有利于并发使用多处理器资源
  - 用户级线程
    - 由用户层调度，减少上下文切换开销
  - 线程模型
    - 一对一模型
    - 多对一模型
    - 多对多模型

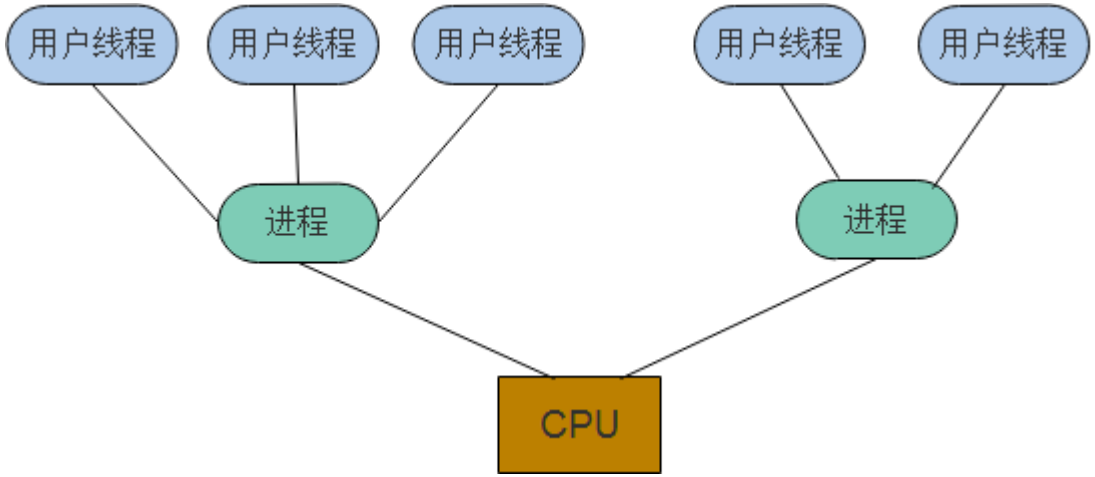
线程模型

- 一对一模型
  - 用户线程通过LWP关联内核线程
  - 线程调度由内核完成
  - SMP、并发使用CPU资源
  - 线程间同步由用户层完成
  - Linux、Windows家族(XP之前)



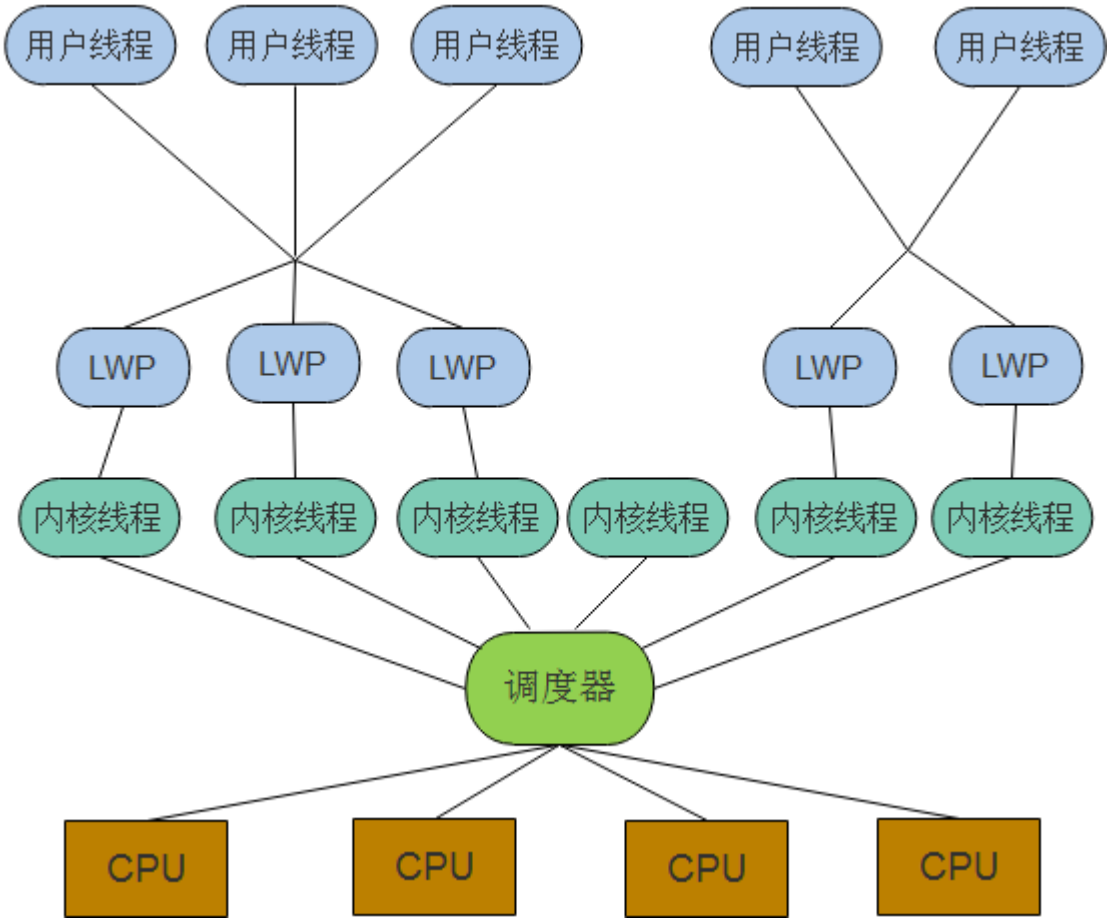
线程模型

- 多对一模型
  - 多个用户线程与一个内核线程关联
  - 线程管理由用户完成、CPU仍以进程为调度单位
  - 单处理器
  - Solaris线程库：Green thread



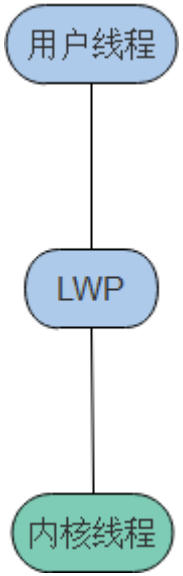
线程模型

- 多对多模型
  - 内核线程为CPU调度单元
  - 用户线程管理



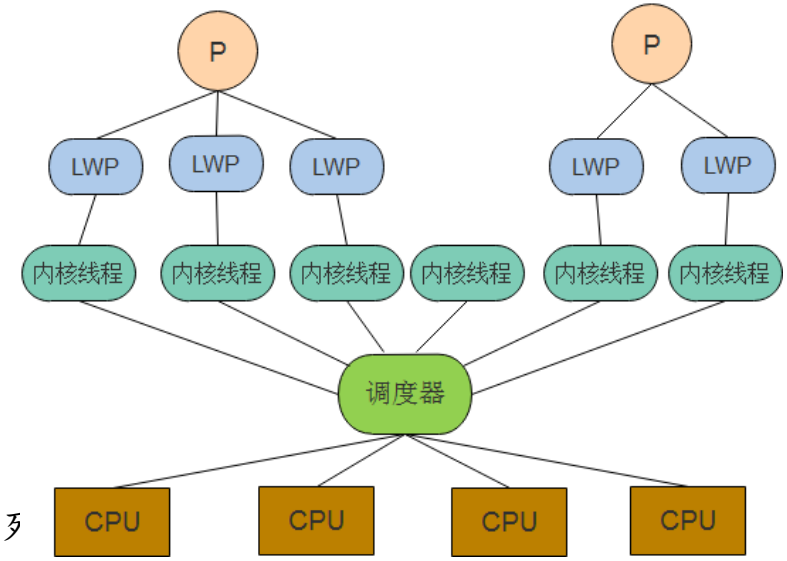
Linux下的线程

- 一对一线程模型
  - 一个轻量进程(LWP)对应一个线程，
  - 每个LWP都与一个内核线程关联
  - 内核线程，通过LWP绑定，调度用户线程
  - 内核线程被阻塞，LWP也阻塞，用户线程也阻塞
  - 调度由内核完成
    - SCHED_OTHER：分时调度策略
    - SCHED_FIFO：实时调度策略：FIFO
    - SCHED_RR：实时调度策略：时间片轮转
  - 创建线程、同步等API由用户线程库完成
    - Linuxthreads：线程PID、信号处理存在不足
    - NPTL(Native POSIX Thread Library)



线程调度与运行

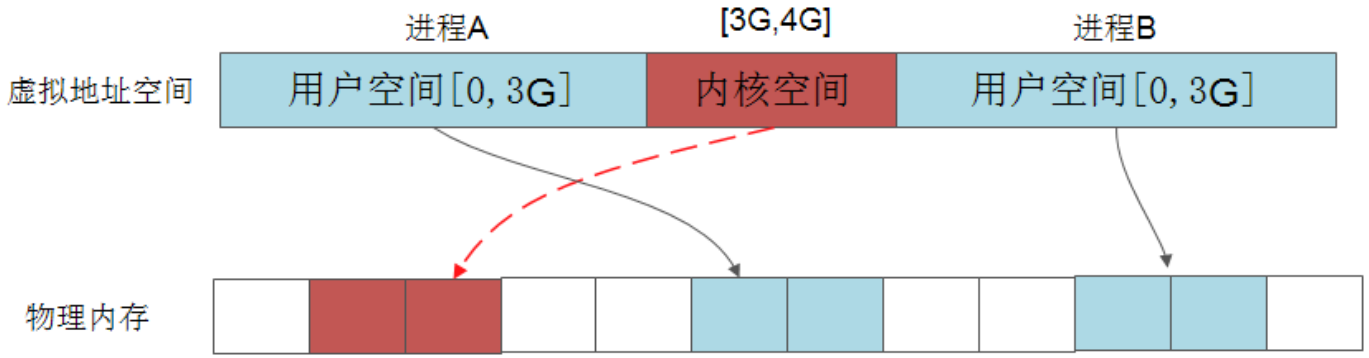
- LWP与普通用户进程比较
  - LWP只有一个最小的执行上下文和调度程序需要的统计信息
  - 用户进程有独立地址空间，LWP与父进程共享地址空间
  - LWP可以像内核线程一样，全局范围内竞争处理器资源
  - LWP调度可以跟用户进程、内核线程一样调度
  - 每一个用户进程可能有一个或多个LWP
  - 通过clone，各进程共享地址空间和资源
    - CLONE_VM、 CLONE_FS
    - CLONE_FILES、CLONE_SIGHAND
  - $ top -H -p <pid>
  - 查看某个指定PID进程下的线程运行

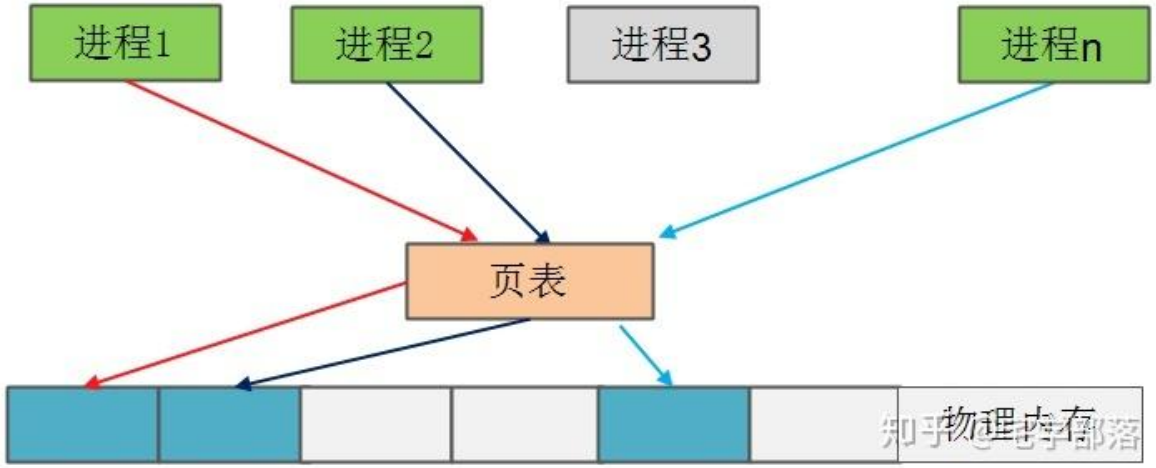


线程安全

进程的原生安全

- 进程的地址空间



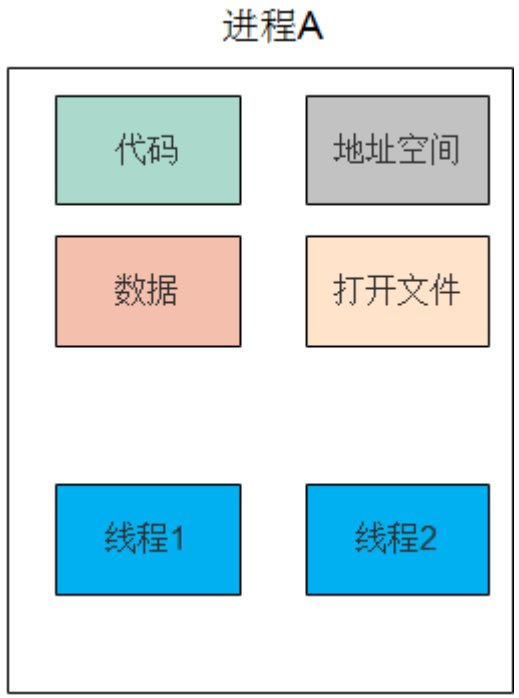


多线程下的资源访问

- 资源划分
  - 共享的资源
    - 代码段、数据段、地址空间
    - 打开的文件、信号处理程序
  - 独占的资源
    - 程序计数器：PC
    - 寄存器
    - 栈空间
      - 不同体系不同分配方式
      - 用户线程和管理线程栈是分离的
  - 进程与线程资源
    - 一室一厅一卫
    - 三室一厅一卫

线程安全

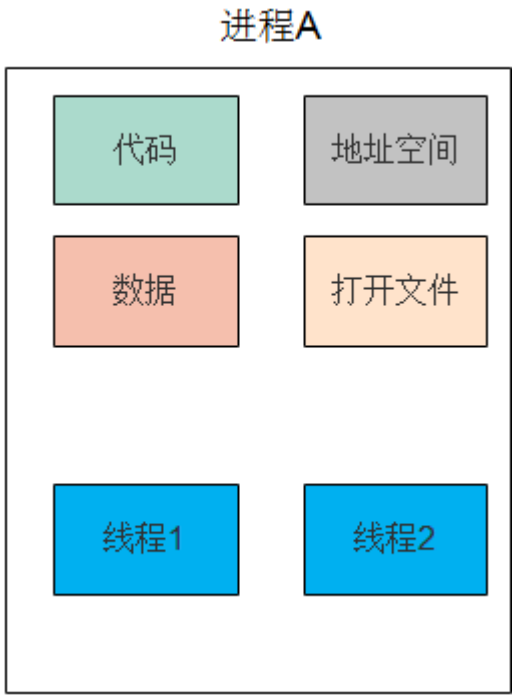
- 对共享资源的安全访问
  - 临界区与临界资源
  - 关中断
  - 锁、条件变量、读写锁
- 函数引用
  - 可重入函数
  - 线程安全函数



线程同步：互斥锁

多线程下的资源访问

- 竞争访问
  - 全局变量
  - 缓冲区
  - 三室一厅卫生间



互斥锁

- 相关API函数
  - int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
  - int pthread_mutex_lock(pthread_mutex_t *mutex);
  - int pthread_mutex_trylock(pthread_mutex_t *mutex);
  - int pthread_mutex_unlock(pthread_mutex_t *mutex);
  - int pthread_mutex_destroy(pthread_mutex_t *mutex);

线程同步：条件变量

条件变量

- 基本概念
  - 互斥锁缺陷：
    - 不断加锁解锁、查询满足条件，开销很大
    - 加锁开销：用户态-内核态-用户态，阻塞在内核态
    - 解锁开销：用户态-内核态-用户态，唤醒等待线程
  - 条件变量
    - 互斥锁(mutex)搭配使用，允许线程阻塞，等待条件满足的信号
  - 优势：
    - 将互斥锁和条件变量绑定
    - 省去了不断加锁解锁的开销
    - 可以使用广播(broadcast)唤醒所有绑定到该条件变量的线程

条件变量

- 相关API函数
  - pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
  - int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *cond_attr);
  - int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
  - int pthread_cond_signal (pthread_cond_t *cond);
  - int pthread_cond_broadcast (pthread_cond_t *cond);
  - int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);
  - int pthread_cond_destroy (pthread_cond_t *cond);

线程同步：条件变量

条件变量

- 基本概念
  - 互斥锁缺陷：
    - 不断加锁解锁、查询满足条件，开销很大
    - 加锁开销：用户态-内核态-用户态，阻塞在内核态
    - 解锁开销：用户态-内核态-用户态，唤醒等待线程
  - 条件变量
    - 互斥锁(mutex)搭配使用，允许线程阻塞，等待条件满足的信号
  - 优势：
    - 将互斥锁和条件变量绑定
    - 省去了不断加锁解锁的开销
    - 可以使用广播(broadcast)唤醒所有绑定到该条件变量的线程

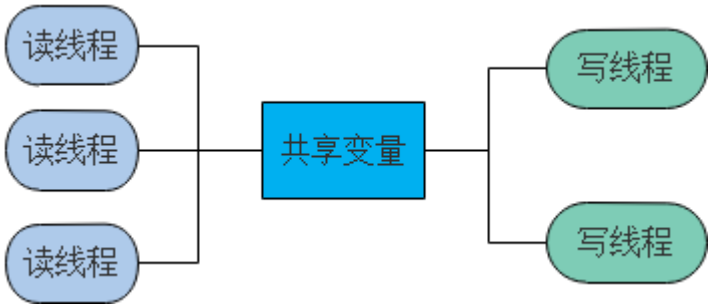
条件变量

- 相关API函数
  - pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
  - int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *cond_attr);
  - int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
  - int pthread_cond_signal (pthread_cond_t *cond);
  - int pthread_cond_broadcast (pthread_cond_t *cond);
  - int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);
  - int pthread_cond_destroy (pthread_cond_t *cond);

线程同步：读写锁

线程同步：读写锁

- 基本概念
  - 互斥锁：同一时刻只允许一个线程读或写
  - 读写锁
    - 允许多个读线程同时读
    - 只允许一个线程写，写的时候会阻塞其它线程(包括读线程)
    - 写优先级高于读



线程同步：读写锁

- 相关API函数
  - pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
  - pthread_rwlock_init (pthread_rwlock_t *restrict rwlock,
    const pthread_rwlockattr_t *restrict attr);
  - int pthread_rwlock_rdlock (pthread_rwlock_t *rwlock);
  - int pthread_rwlock_wrlock (pthread_rwlock_t *rwlock);
  - int pthread_rwlock_tryrdlock (pthread_rwlock_t *rwlock);
  - int pthread_rwlock_trywrlock (pthread_rwlock_t *rwlock);
  - int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);
  - int pthread_rwlock_destroy (pthread_rwlock_t *rwlock);

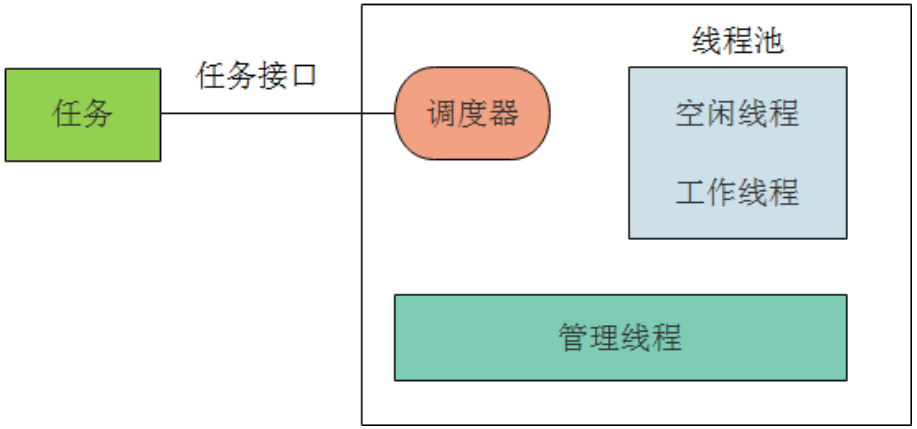
线程池的概念

线程池的概念

- 线程的开销
  - 系统调用的开销：线程的创建、销毁
  - 上下文切换、互斥锁等加锁解锁
- 线程池原理
  - 预先在池中创建一些线程
  - 无任务时，线程阻塞在池中
  - 有任务时，将任务分配到指定的线程执行
  - 池中线程的数目甚至可根据任务多少动态删减

线程池的概念

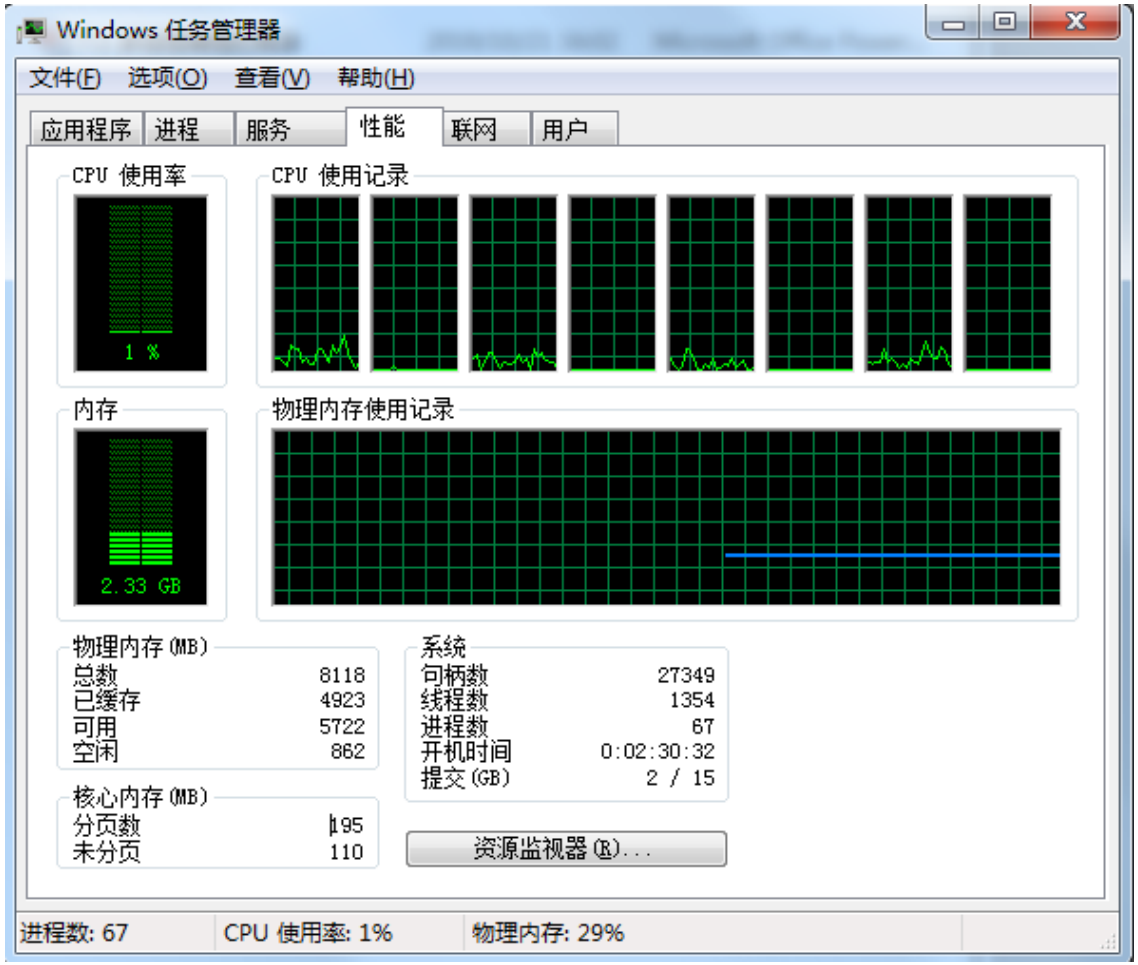
- 实现原理
  - 管理线程
    - 创建并管理线程
    - 任务分配运行
  - 工作线程
    - 线程池中实际执行任务的线程
  - 任务接口
    - 每个任务的实现接口



超线程技术

四核八线程

- 任务管理器


Windows 任务管理器
文件(F) 选项(O) 查看(V) 帮助(H)
应用程序 进程 服务 性能 联网 用户
CPU 使用率
1 %
CPU 使用记录
内存
2.33 GB
物理内存使用记录
物理内存(MB)
总数 8118
已缓存 4923
可用 5722
空闲 862
核心内存(MB)
分页数 195
未分页 110
系统
句柄数 27349
线程数 1354
进程数 67
开机时间 0:02:30:32
提交(GB) 2 / 15
资源监视器(R)...
进程数: 67
CPU 使用率: 1%
物理内存: 29%

超线程技术

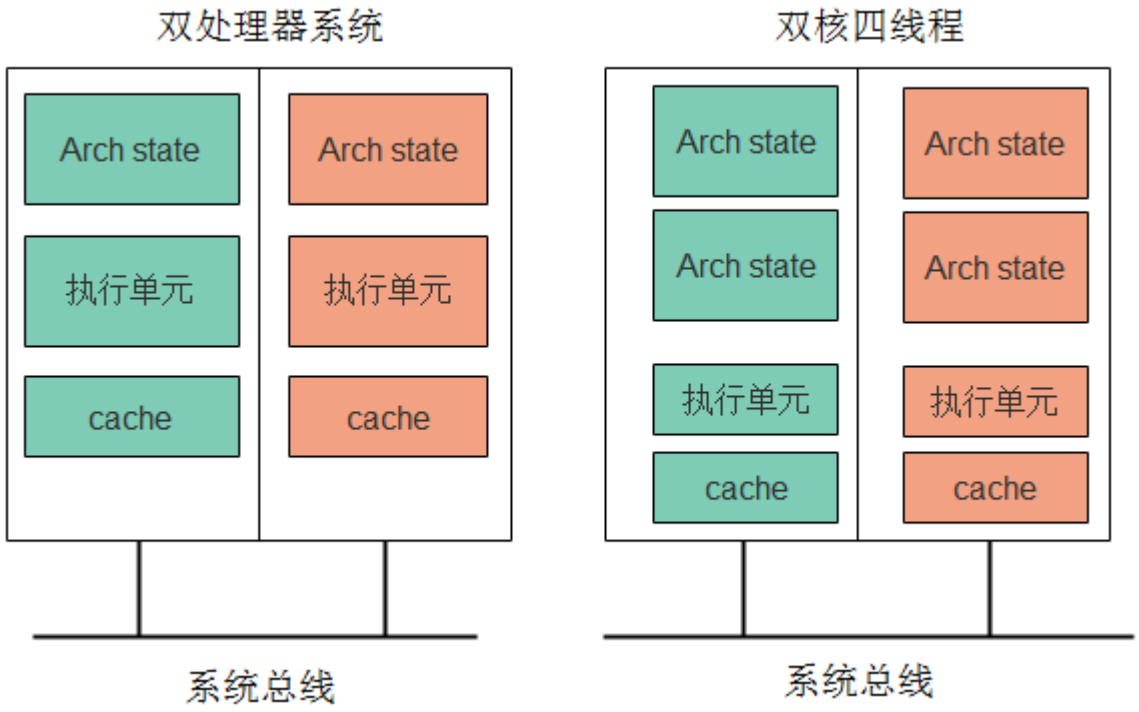
- 基本概念
  - 共享打印机
    - 电脑---打印机
    - 电脑1---打印机
    - 电脑2---打印机
  - 超线程技术
    - Hyper-Threading，简称HT
    - 使用特殊指令将一个物理处理器视为2个逻辑处理器
    - 每个逻辑处理器都可以分配一个线程运行
    - 最大限度地提升CPU资源利用率

超线程技术

- 实现原理
  - CPU微架构资源划分
    - 复制的资源：每个逻辑CPU都有一套完整的体系结构状态
    - 划分的资源：重定序(re-order)缓冲、load/store缓冲、队列等
      多任务模式下这些资源划分给2个逻辑CPU使用
      单任务模式下这些资源划分给1个逻辑CPU使用
    - 共享的资源：执行计算单元(加、减)、cache、总线
  - 主板的支持
    - 如I7的X58芯片组
    - 主板BIOS支持超线程
    - 操作系统的支持：Windows XP以后、Linux2.4以后
    - 应用层支持：NPTL库

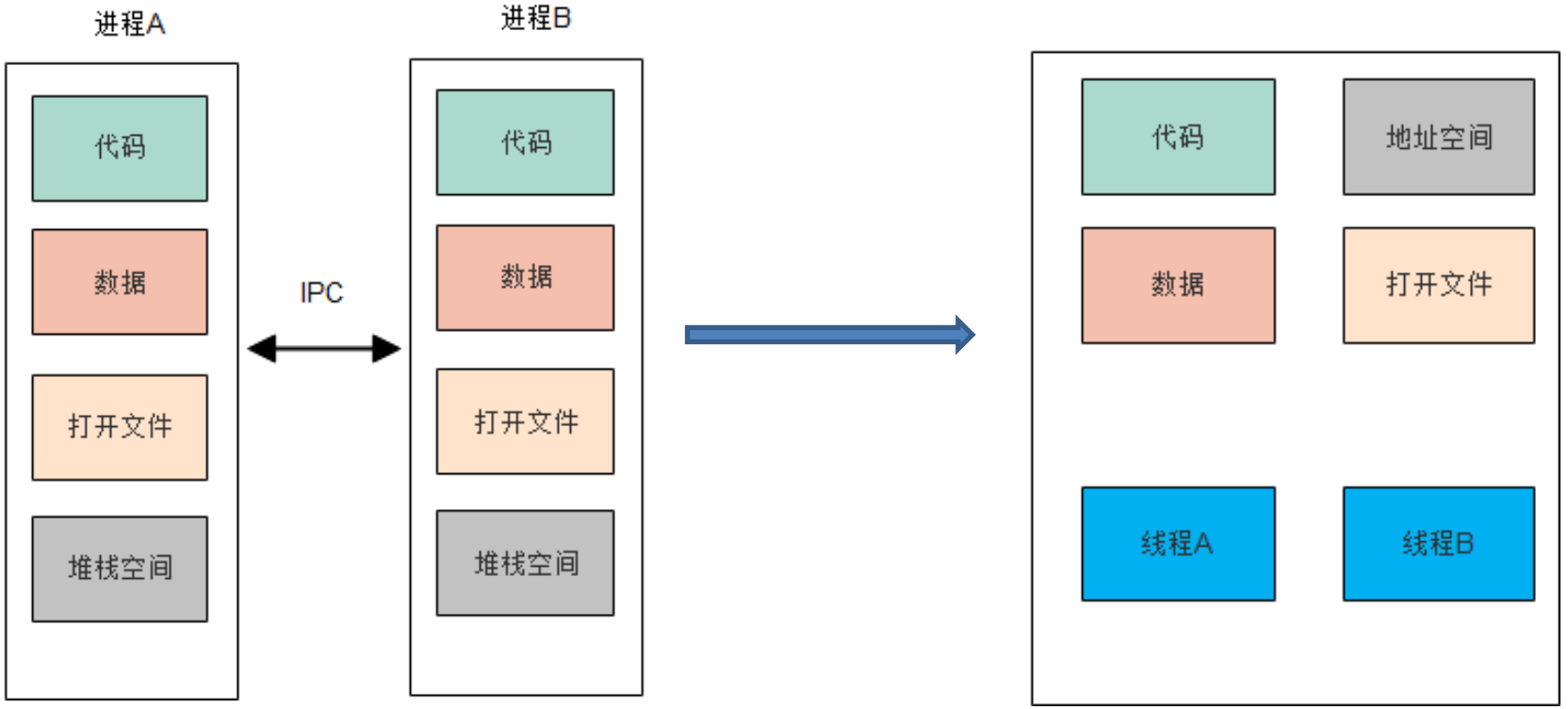
超线程技术

- 实现原理
  - 交替工作
  - 共享单元
    - 执行单元
    - 缓存
    - 总线
  - 应用场所
    - 服务器
    - 工作站



协程的概念

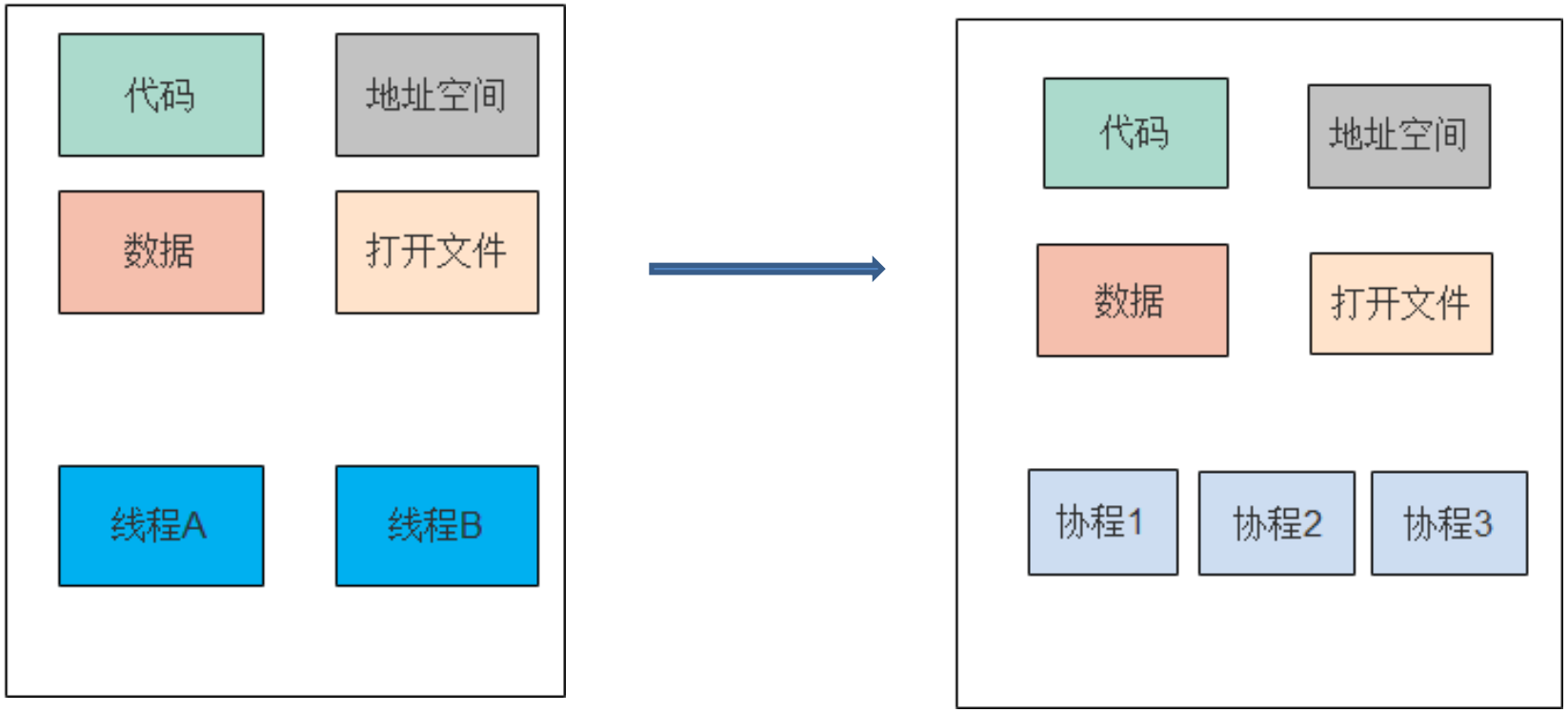
从进程到线程


进程A
代码
数据
打开文件
堆栈空间
IPC
进程B
代码
数据
打开文件
堆栈空间
代码
地址空间
数据
打开文件
线程A
线程B

协程的概念

- 基本概念
  - 线程的开销
    - 上下文切换开销
    - 互斥锁的开销
  - 协程
    - 对共享资源的访问由程序本身维护控制
    - 不需要锁，无调度成本，执行效率高
    - 适合彼此熟系的程序组件：合作式多任务、管道
    - 进程 + 线程 → 进程 + 协程

从线程到协程



使用协程编程

- 协程的实现
  - Python：yield/send、async/await(python3.5以后)
  - Lua：Lua5.0开始使协程
  - Go：支持线程，后端并发
  - C语言：协程库

小结

- 进程、线程与协程
  - 进程、线程是系统级，而协程是语言级
  - 进程：资源分配的基本单位
  - 线程：程序执行的基本单位
  - 切换成本：进程 > 线程 > 协程
  - 安全性：进程 > 线程 > 协程
  - 协程缺陷：无法利用多核CPU，做到真正的并发