IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

操作系统

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

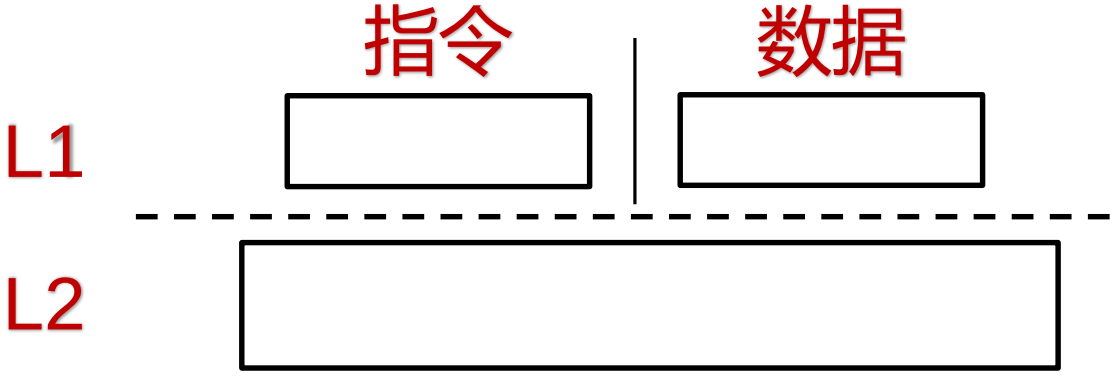
https://ipads.se.sjtu.edu.cn

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

回顾：TLB结构简介

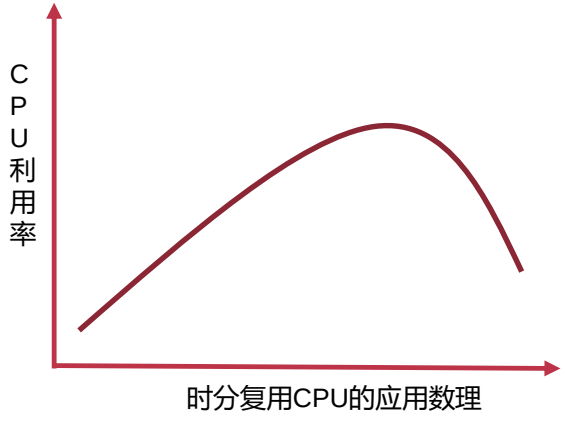
- 回顾：ICS中学习的分级cache结构 （L1/L2/L3）
- TLB设计通常也采用分级结构 （以AARCH64为例）



- **思考：为什么采用分级结构?**

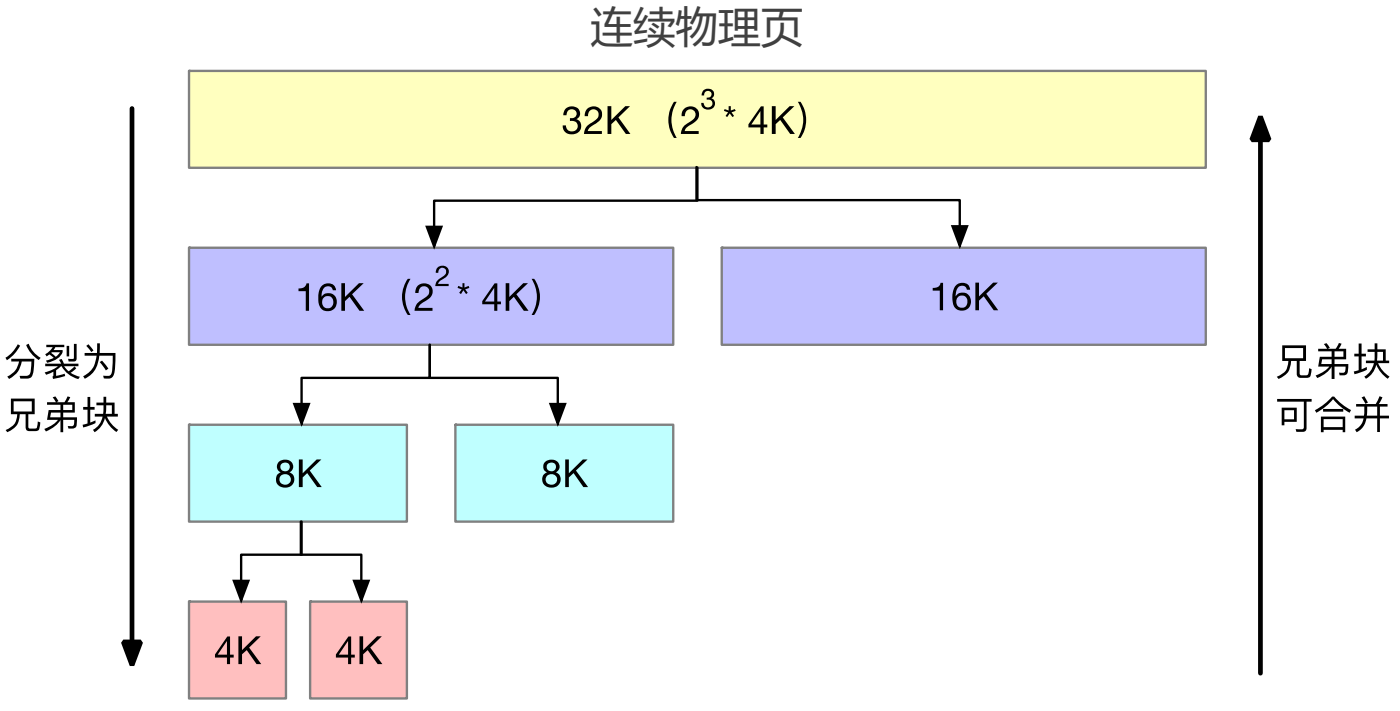
回顾：Thrashing Problem

- **直接原因**
  - 过于频繁的缺页异常（物理内存总需求过大）
- **大部分 CPU 时间都被用来处理缺页异常**
  - 等待缓慢的磁盘 I/O 操作
  - 仅剩小部分的时间用于执行真正有意义的工作
- **调度器造成问题加剧**
  - 等待磁盘 I/O导致CPU利用率下降
  - 调度器载入更多的进程以期提高CPU利用率
  - 触发更多的缺页异常、进一步降低CPU利用率、导致连锁反应



回顾：物理内存管理之buddy system

- 伙伴系统 （能避免外部碎片吗？）



进程

进程的诞生和概念 – 进程的状态 – 数据结构 – 基本操作

再回来看Hello World

```c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

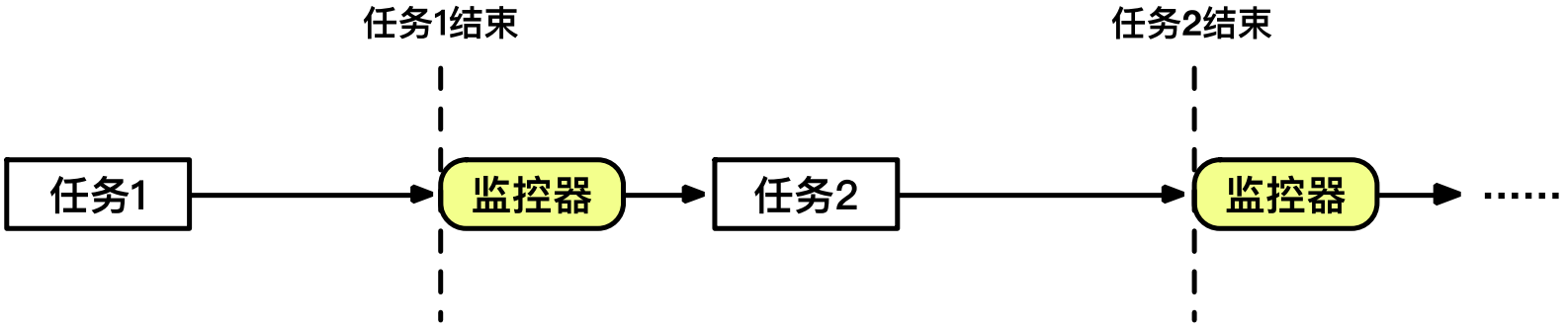
```
bash$ gcc hello.c -o hello

# 同时启动两个hello world程序
bash$ ./hello & ./hello
[1] 144
Hello World!
Hello World!
[1]+ Done       ./hello
```

运行多个hello时，操作系统怎么抽象与管理?

进程的诞生：从单任务到多任务

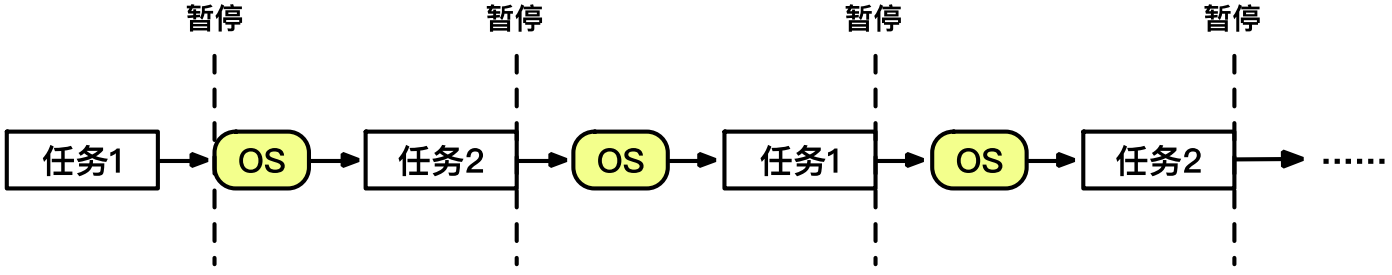
- **早期的计算机一次只能执行一个任务**



- **计算机的发展趋势**
  - 计算机程序种类越来越多（文本编辑、科学计算、web服务……）
  - 外部设备种类越来越多（硬盘、显示器、网络），造成程序等待

进程的诞生：从单任务到多任务

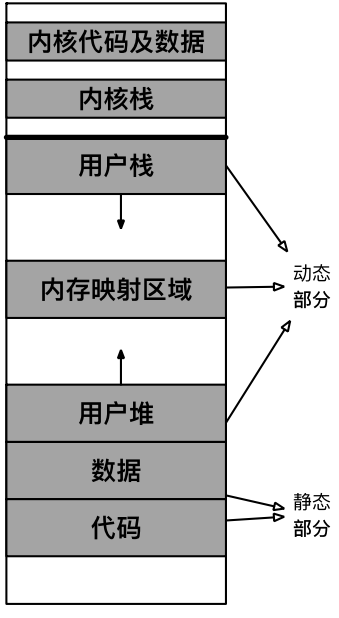
- **思路：提出分时（time-sharing）操作系统**
  - 多任务并行：当一个任务需要等待时，切换到其他任务



- **任务的抽象——进程**
  - 进程的执行**状态**不断更新
  - 不断切换处理器上运行的进程（**上下文切换**）
  - 操作系统需要对进程进行**调度**（**且听下回分解**）

进程：运行中的程序

- **进程是计算机程序运行时的抽象**
  - 静态部分：程序运行需要的代码和数据
  - 动态部分：程序运行期间的状态（程序计数器、堆、栈……）
- **进程具有独立的虚拟地址空间**
  - 每个进程都具有“独占全部内存”的假象
  - 内核中同样包含内核栈和内核代码、数据



简化模型：单一线程的进程

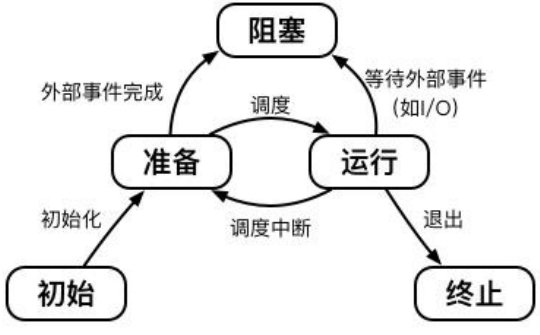
- **假设每个进程都只有一个线程**
  - 历史上确实如此！（如早期的UNIX操作系统）

- **好处：便于理解操作系统中的各种概念**
  - 多线程使操作系统的管理更加复杂
  - 在单一线程的进程中：线程管理 / 调度 ≈ 进程管理 / 调度
  - 线程的内容将在后面介绍

进程的状态

- **进程至少应当拥有以下五种状态：**
  - 新生状态（new）：进程刚被创建
  - 运行状态（running）：进程正在处理器上运行
  - 准备状态（ready）：进程可以运行，但没有被调度
  - 阻塞状态（blocked）：进程进入等待状态，短时间不再运行
  - 终结状态（terminated）：进程完成了执行

- **进程会不断进行状态切换**
  - 被调度器调度，开始执行： 准备->运行



进程的相关数据结构：Process Control Block

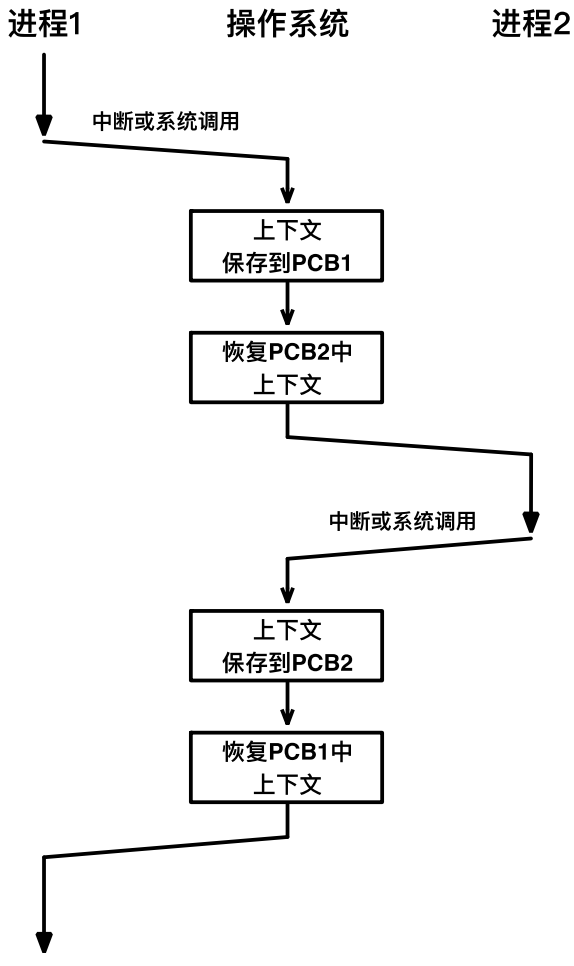
- **存放进程相关的各种信息**
  - 进程的标识符、内存、打开的文件……
  - 进程在切换时的状态（上下文context）

```
label_t u_qsav;          /* label variable for quits and interrupts */
label_t u_ssav;          /* label variable for swapping */
int u_signal[NSIG];      /* disposition of signals */
time_t  u_utime;         /* this process user time */
time_t  u_stime;         /* this process system time */
time_t  u_cutime;        /* sum of childs' utimes */
time_t  u_cstime;        /* sum of childs' stimes */
int *u_ar0;          /* address of users saved R0 */
```

UNIX v7的部分PCB （u_ar0为上下文信息）

进程的上下文切换

- **进程通过中断或系统调用进入内核**
- **上下文保存在对应的PCB中**
  - 被调度时，从PCB中取出上下文并恢复

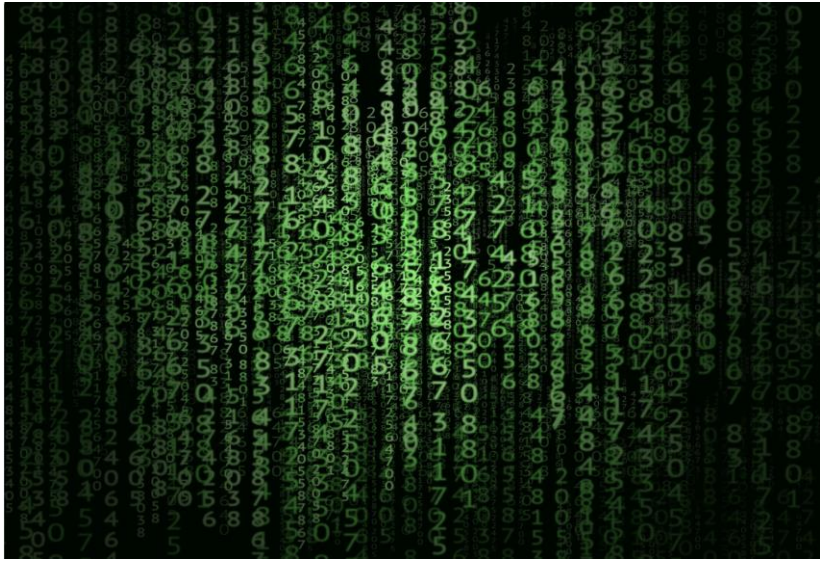


进程的基本操作接口

- **进程创建：fork (spawn, vfork, clone)**
- **进程执行：exec**
- **进程间同步：wait**
- **进程退出：exit/abort**

……

操作系统 vs《The Matrix/黑客帝国》



The Matrix	操作系统
民众	进程
Oracle/先知	进程调度器
列车员	进程间通信
Key Maker	秘钥管理程序
特工/Smith	杀毒程序（后来演变为病毒）
电话	系统调用
建筑师	内核监控程序

https://www.techug.com/post/jul-how-do-the-matrix-as-an-operating-system.html

进程创建：fork()

- **语义：为调用进程创建一个一模一样的新进程**
  - 调用进程为**父进程**，新进程为**子进程**
  - 接口简单，无需任何参数

- **fork后的两个进程均为独立进程**
  - 拥有不同的进程id
  - 可以并行执行，互不干扰（除非使用特定的接口）
  - 父进程和子进程会共享部分数据结构（内存、文件等）

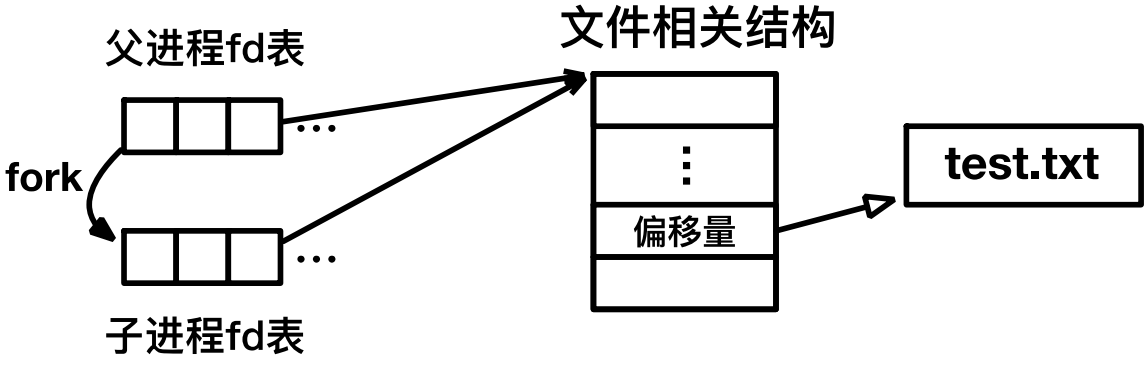
fork的示例

- **考虑以下代码**
  - 假设先执行父进程，然后再执行子进程
  - test.txt中的内容：”abcdefghijklmnopqrst…”

```
char str[10];
int fd = open("test.txt", O_RDWR);
if (fork() == 0) {
    ssize_t cnt = read(fd, str, 10);
    printf("Child process: %s\n", (char *)str);
} else {
    ssize_t cnt = read(fd, str, 10);
    printf("Parent process: %s\n", (char *)str);
}
```

fork的示例

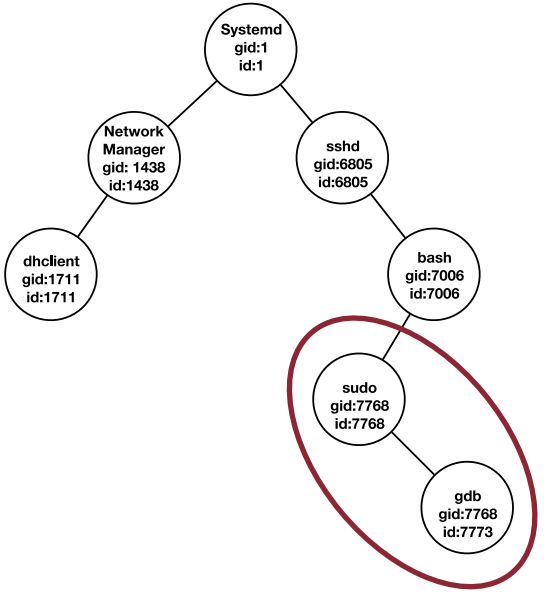
- **执行结果(如果parent先执行):**
  - `Parent process: abcdefghij`
  - `Child process: klmnopqrst`
  - 原因：两个进程共享了同一个指向文件的结构体



进程树与进程组

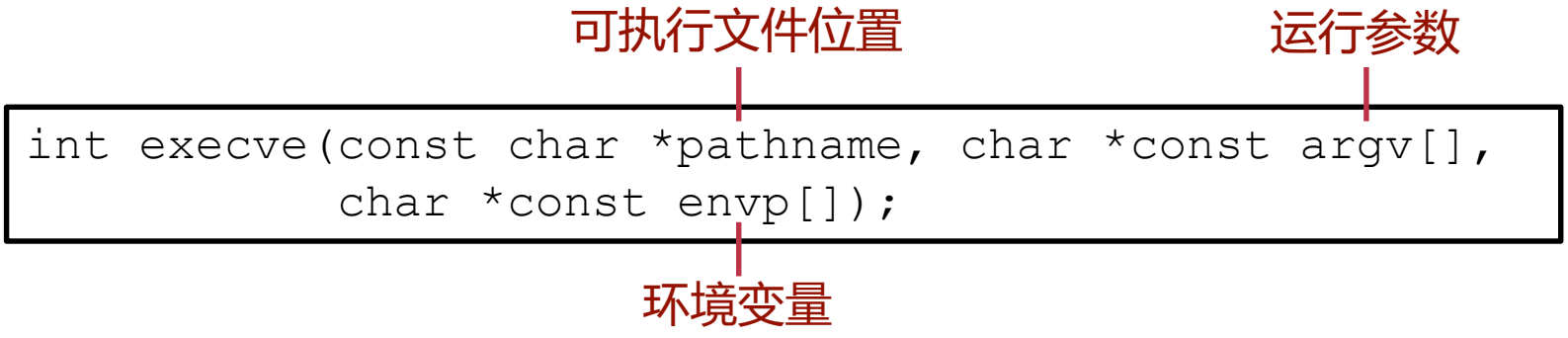
- **fork为进程之间建立了父进程和子进程的关系**
  - 进程之间建立了树型结构
  - Linux可使用ps命令查看

- **多个进程可以属于同一个进程组**
  - 子进程默认与父进程属于同一个进程组
  - 可以向同一进程组中的所有进程发送信号
  - 主要用于shell程序中



进程的执行：exec

- **为进程指定可执行文件和参数**



- **在fork之后调用**
  - exec在载入可执行文件后会重置地址空间

写时拷贝（Copy-On-Write）

- **早期的fork实现：将父进程直接拷贝一份**
  - 性能差：时间随占用内存增加而增加
  - 无用功：fork之后如果调用exec，拷贝的内存就作废了

- **基本思路：只拷贝内存映射，不拷贝实际内存**
  - 性能较好：一条映射至少对应一个4K的页面
  - 调用exec的情况里，减少了无用的拷贝

fork的优缺点分析

- **fork的优点**
  - 接口非常简洁
  - 将进程“创建”和“执行”（exec）解耦，提高了灵活度
  - 刻画了进程之间的内在关系（进程树、进程组）
- **fork的缺点**
  - 完全拷贝过于粗暴（不如clone）
  - 性能差、可扩展性差（不如vfork和spawn）
  - 不可组合性 (例如：fork() + pthread())

fork的替代接口

- **vfork：类似于fork，但让父子进程共享同一地址空间**
  - 优点：连映射都不需要拷贝，性能更好
  - 缺点：
    - 只能用在”fork + exec”的场景中
    - 共享地址空间存在安全问题
- **轶事：vfork的提出最初就是为了解决fork的性能问题**
  - 但写时拷贝拯救了fork

> Since this function is hard to use correctly from application software, it is recommended to use posix_spawn(3) or fork(2) instead.

fork的替代接口

- **posix_spawn: 相当于fork + exec**
  - 优点：可扩展性、性能较好
  - 缺点：不如fork灵活

- **clone: fork的“进阶版”，可以选择性地不拷贝内存**
  - 优点：高度可控，可依照需求调整
  - 缺点：接口比fork复杂，选择性拷贝容易出错

线程

线程的概念 - 线程模型 - 相关数据结构 - 基本操作 - 上下文切换

为什么需要线程?

- **创建进程的开销较大**
  - 包括了数据、代码、堆、栈等
- **进程的隔离性过强**
  - 进程间交互：可以通过进程间通信（IPC），但开销较大
- **进程内部无法支持并行**

线程：更加轻量级的运行时抽象

- **线程只包含运行时的状态**
  - 静态部分由**进程**提供
  - 包括了执行所需的**最小**状态（主要是寄存器和栈）

- **一个进程可以包含多个线程**
  - 每个线程共享同一地址空间（方便数据共享和交互）
  - 允许进程内并行

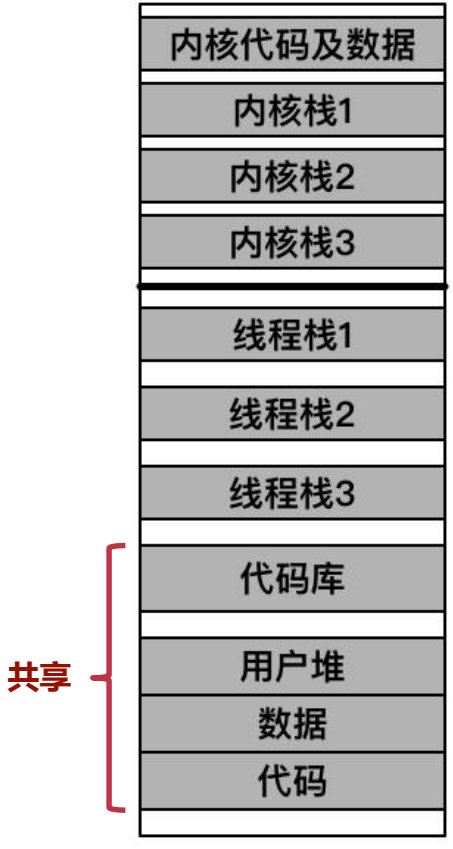
进阶模型：多线程的进程

- **一个进程可以包含多个线程**

- **一个进程的多线程可以在不同处理器上同时执行**
  - 调度的基本单元由进程变为了线程
  - 每个线程都有**状态**
  - 上下文切换的单位变为了线程

多线程进程的地址空间

- 每个线程拥有自己的栈
- 内核中也有为线程准备的内核栈
- 其它区域共享
  - 数据、代码、堆……

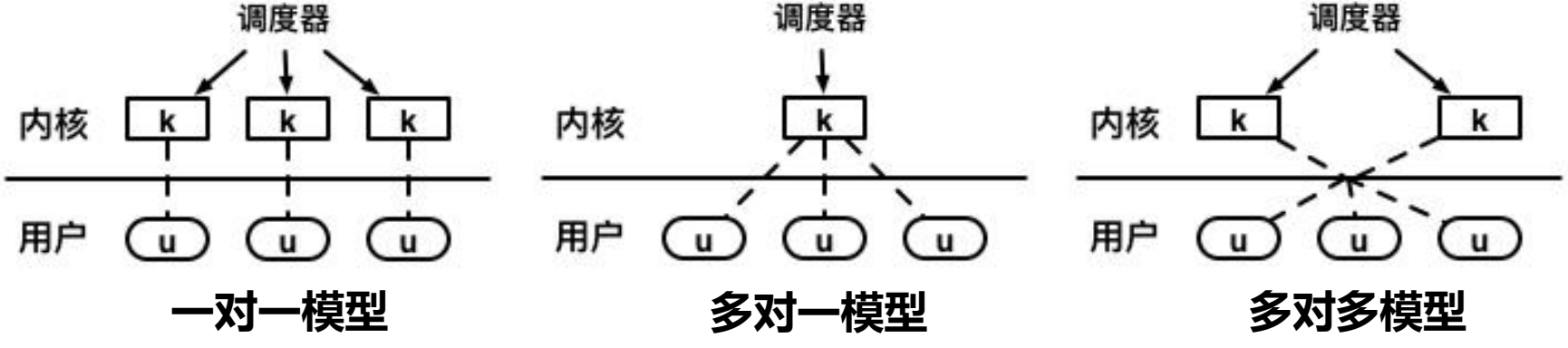


用户态线程与内核态线程

- **根据线程是否受内核管理，可以将线程分为两类**
  - 内核态线程：内核可见，受内核管理
  - 用户态线程：内核不可见，不受内核直接管理
- **内核态线程**
  - 由内核创建，线程相关信息存放在内核中
- **用户态线程（纤程）**
  - 在应用态创建，线程相关信息主要存放在应用数据中

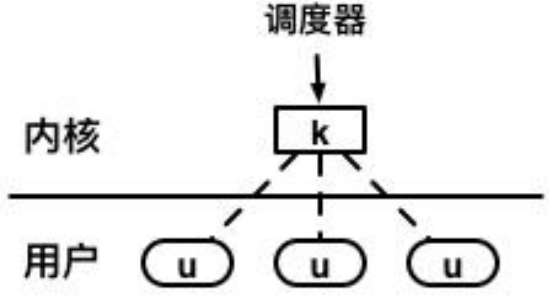
线程模型

- **线程模型表示了用户态线程与内核态线程之间的联系**
  - 多对一模型：多个用户态线程对应一个内核态线程
  - 一对一模型：一个用户态线程对应一个内核态线程
  - 多对多模型：多个用户态线程对应多个内核态线程



多对一模型

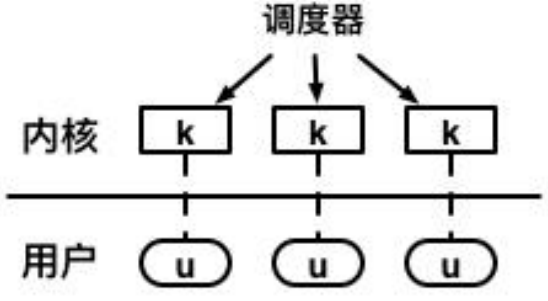
- **将多个用户态线程映射给单一的内核线程**
  - 优点：内核管理简单
  - 缺点：可扩展性差，无法适应多核机器的发展
- **在主流操作系统中被弃用**
- **用于各种用户态线程库中**



一对一模型

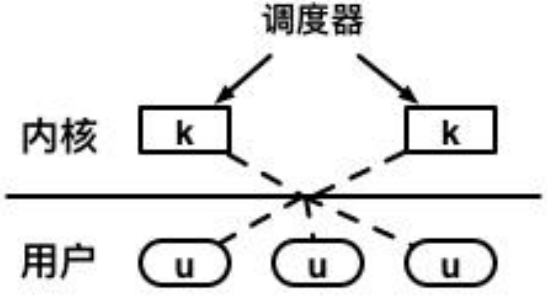
- **每个用户线程映射单独的内核线程**
  - 优点：解决了多对一模型中的可扩展性问题
  - 缺点：内核线程数量大，开销大

- **主流操作系统都采用一对一模型**
  - Windows、Linux、OS X……



多对多模型（又叫Scheduler Activation）

- **N个用户态线程映射到M个内核态线程（N > M）**
  - 优点：解决了可扩展性问题（多对一）和线程过多问题（一对一）
  - 缺点：管理更为复杂
- **Solaris在9之前使用该模型**
  - 9之后改为一对一
- **在虚拟化中得到了广泛应用**



线程的相关数据结构：TCB

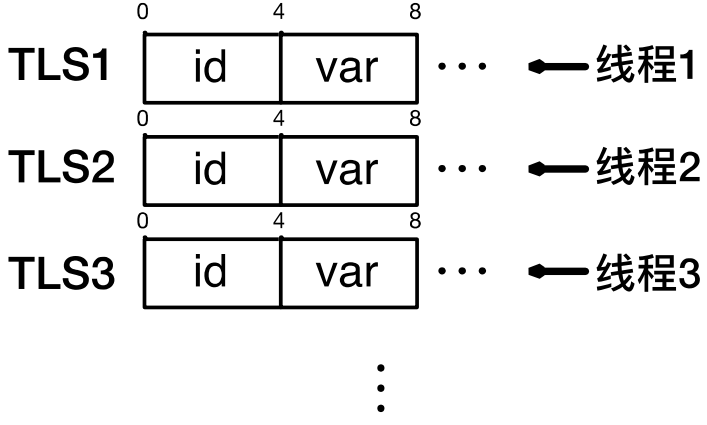
- **一对一模型的TCB可以分为两部分**
- **内核态：与PCB结构类似**
  - Linux中进程与线程使用的是同一种数据结构（task_struct）
  - 上下文切换中会使用
- **应用态：可以由线程库定义**
  - Linux：pthread结构体
  - Windows：TIB（Thread Information Block）
  - 可以认为是内核TCB的扩展

线程本地存储（TLS）

- **不同线程可能会执行相同的代码**
  - 线程不具有独立的地址空间，多线程共享代码段
- **问题：对于全局变量，不同线程可能需要不同的拷贝**
  - 举例：用于标明系统调用错误的errno
- **解决方案：线程本地存储（Thread Local Storage）**

线程本地存储（TLS）

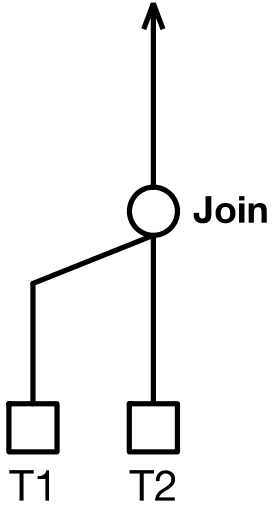
- **线程库允许定义每个线程独有的数据**
  - `__thread int id;` 会为每个线程定义一个独有的id变量
- **每个线程的TLS结构相似**
  - 可通过TCB索引
- **TLS寻址模式：基地址 + 偏移量**
  - X86: 段页式 (fs寄存器)
  - AArch64: 特殊寄存器tpidr_el0



线程的基本操作：以*pthreads*为例

- **创建：pthread_create**
  - 内核态：创建相应的内核态线程及内核栈
  - 应用态：创建TCB、应用栈和TLS

- **合并：pthread_join**
  - 等待另一线程执行完成，并获取其执行结果
  - 可以认为是fork的“逆向操作”



线程的基本操作：以*pthreads*为例

- **退出：pthread_exit**
  - 可设置返回值（会被pthread_join获取）

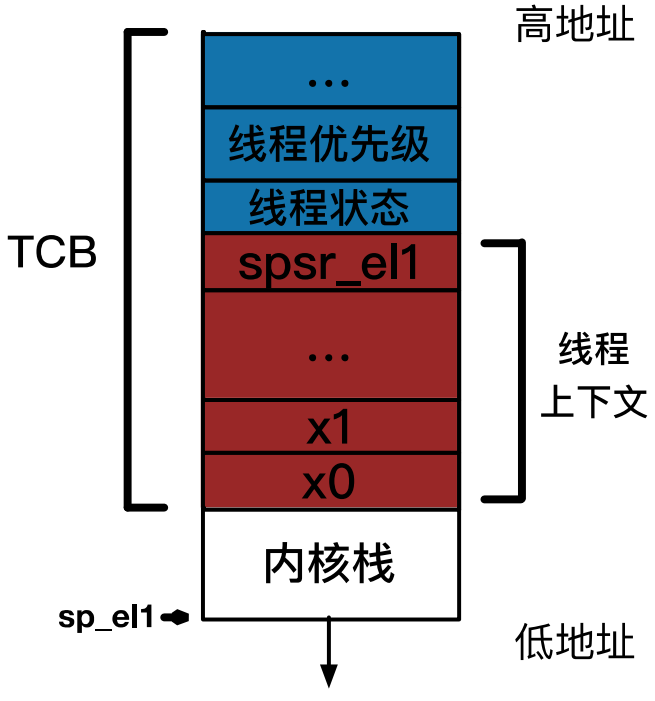
- **暂停：pthread_yield**
  - 立即暂停执行，出让CPU资源给其它线程
  - 好处：可以帮助调度器做出更优的决策

线程的上下文切换：以*ChCore*为例

- **线程的上下文即重要的寄存器信息**
  - 常规寄存器：x0-x30
  - 程序计数器（PC）： elr_el1
  - 栈指针：sp_el0
  - CPU状态（如条件码）：spsr_el1
- **主要分为三步**
  - 进入内核态，保存上下文
  - 切换页表与内核栈
  - 恢复上下文，返回用户态

ChCore的TCB结构

- **上半部分：线程的相关信息**
- **下半部分：线程上下文**
- **TCB下面为线程的内核栈**
  - 刚进入内核时的线程内核栈为空
  - sp_el1指向栈顶



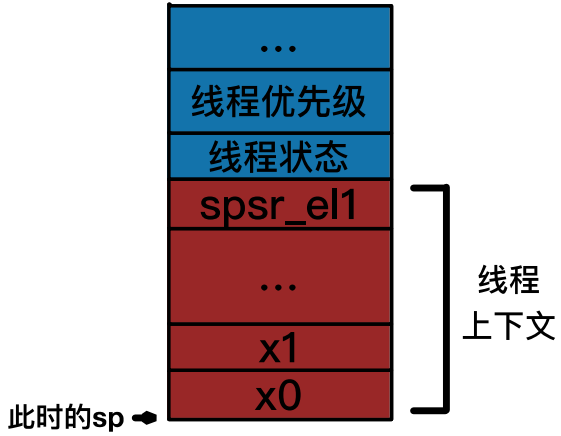
第一步：进入内核态、保存上下文

- **应用线程可通过异常、中断或系统调用进入内核态**
  - 运行状态将切换到内核态（EL1）
  - 开始使用sp_el1作为栈指针（用户栈切换到内核栈）
  - 保存应用线程的PC（elr_el1）
  - 保存应用线程的CPU状态（spsr_el1）
  - 以上均由硬件自动完成

第一步：进入内核态、保存上下文

- 保存上下文

```
sub sp, sp, #ARCH_EXEC_CONT_SIZE
// 保存常规寄存器(x0-x29)
stp x0, x1, [sp, #16 * 0]
stp x2, x3, [sp, #16 * 1]
stp x4, x5, [sp, #16 * 2]
// ...
stp x28, x29, [sp, #16 * 14]
// 保存x30和三个特殊寄存器：sp_el0, elr_e1, spsr_el1
mrs x21, sp_el0
mrs x22, elr_el1
mrs x23, spsr_el1
stp x30, x21, [sp, #16 * 15]
stp x22, x23, [sp, #16 * 16]
```



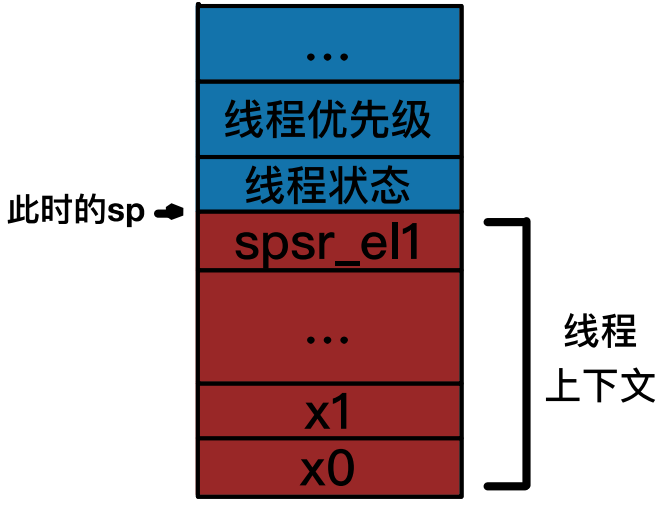
第二步：切换页表和内核栈

- **操作系统确定下一个被调度的线程（调度器决定）**
- **切换页表**
  - 将页表相关寄存器的值置为目标线程的页表基地址
- **切换内核栈**
  - 找到目标内核栈的栈顶指针（目标线程的TCB）
  - 修改sp_el1的值至目标内核栈
  - 可以认为是线程执行的分界点（切换之后变为目标线程执行）

第三步：上下文恢复，返回用户态

- 上下文恢复：取出栈上的值并存回寄存器

```
ldp x22, x23, [sp, #16 * 16]
ldp x30, x21, [sp, #16 * 15]
// 恢复三个特殊寄存器
msr sp_el0, x21
msr elr_el1, x22
msr spsr_el1, x23
// 恢复常规寄存器X0-X29
ldp x0, x1, [sp, #16 * 0]
// ......
ldp x28, x29, [sp, #16 * 14]
add sp, sp, #ARCH_EXEC_CONT_SIZE
```

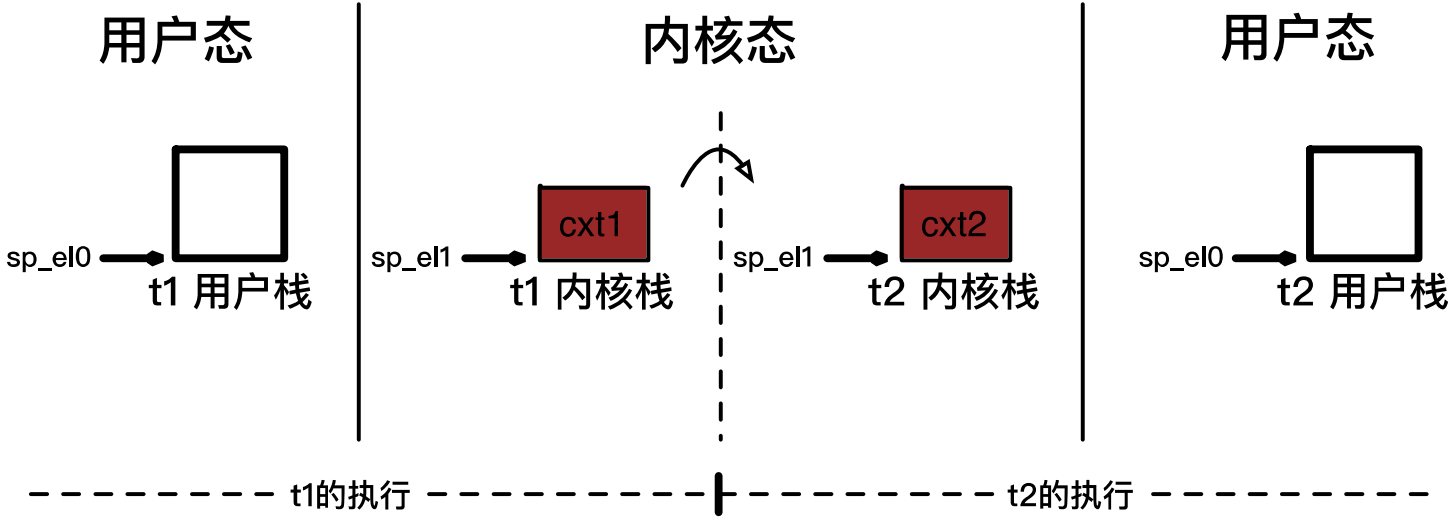


第三步：上下文恢复，返回用户态

- **返回用户态：调用eret，由硬件执行一系列操作**
  - 将elr_el1中的返回地址存回PC
  - 改为使用sp_el0作为栈指针（内核栈切换到用户栈）
  - 将CPU状态设为spsr_el1中的值
  - 运行状态切换为用户态（EL0）

上下文切换小结

- 共涉及两次权限等级切换、三次栈切换
- 内核栈的切换是线程切换执行的“分界点”



纤程

纤程的概念 - 编程模型 - Windows和编程语言支持

一对一线程模型的局限

- **复杂应用：对调度存在更多需求**
  - 生产者消费者模型：生产者完成后，消费者最好马上被调度
  - 内核调度器的信息不足，无法完成及时调度

- **“短命”线程：执行时间亚毫秒级（如处理web请求）**
  - 内核线程初始化时间较长，造成执行开销
  - 线程上下文切换频繁，开销较大

纤程（用户态线程）

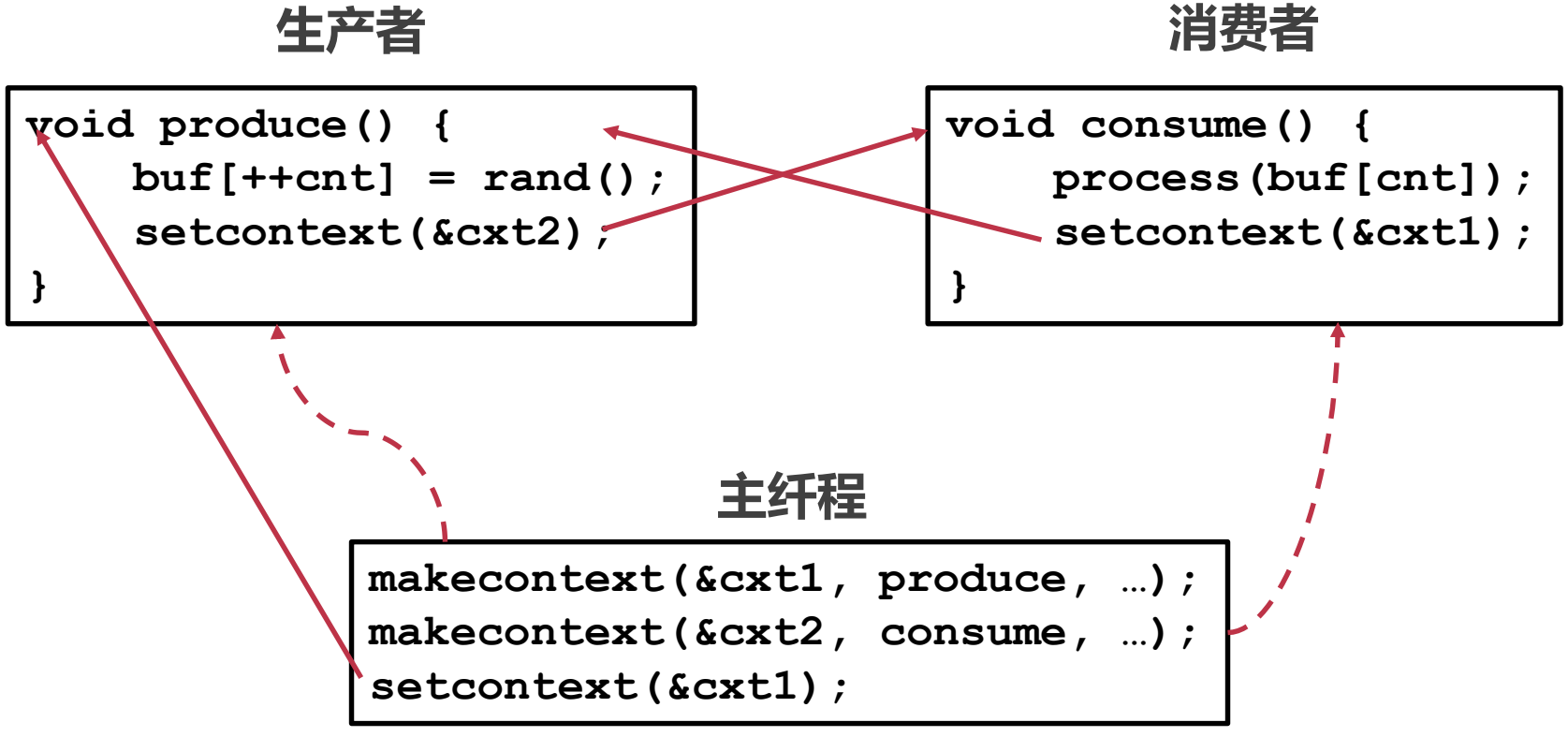
- **比线程更加轻量级的运行时抽象**
  - 不单独对应内核线程
  - 一个内核线程可以对应多个纤程（多对一）

- **纤程的优点**
  - 不需要创建内核线程，开销小
  - 上下文切换快（不需要进入内核）
  - 允许用户态自主调度，有助于做出更优的调度决策

Linux对于纤程的支持：ucontext

- **每个ucontext可以看作一个用户态线程**
  - makecontext: 创建新的ucontext
  - setcontext:纤程上下文切换
  - getcontext:保存当前的ucontext

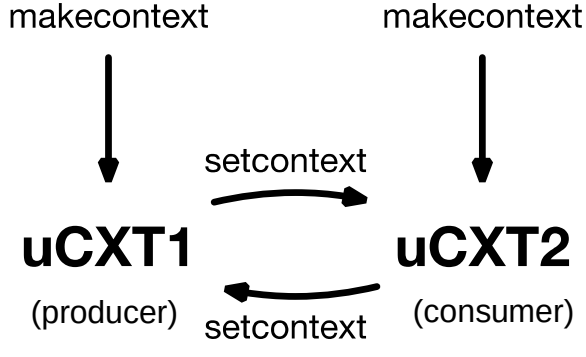
纤程的例子：生产者 - 消费者



从例子看纤程的优势

- **纤程切换及时**
  - 当生产者完成任务后，可直接用户态切换到消费者
  - 对该线程来说是最优调度（内核调度器和难做到）

- **高效上下文切换**
  - 切换不进入内核态，开销小
  - 即时频繁切换也不会造成过大开销



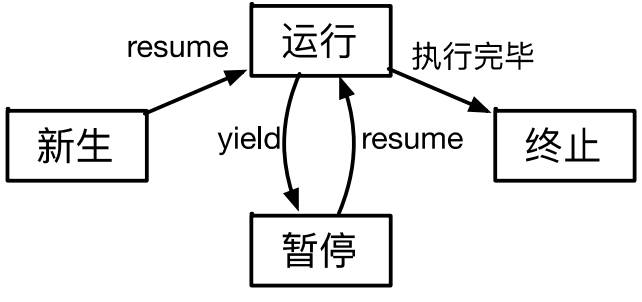
Windows对于纤程的支持：Fiber库

- **与ucontext类似的编程模型**
  - createFiber: 创建新的纤程
  - SwitchToFiber: 纤程切换

- **支持纤程本地存储（FLS）**
  - 当一个内核线程对应单个纤程时，FLS与TLS结构相同
  - 当一个内核线程对应多个纤程时，TLS可分裂为多个FLS

程序语言中对纤程的支持：协程

- **许多高级程序语言都对协程提供了支持**
  - go、python、lua……
  - C++自20开始也支持了协程
- **协程也拥有状态（新生 / 暂停 / 终止 / 执行）**
  - 核心操作：yield（使协程暂停执行）、resume（继续执行）



下次课内容

- **进程 / 线程调度**