Lecture7 Process&Thread

进程

进程：运行中的程序

- 进程是计算机程序运行的抽象
    1. 静态部分：程序运行需要的代码和数据
    2. 动态部分：程序运行期间的状态(PC、stack、heap)
- 进程具有独立的虚拟地址空间
    1. 每个进程都具有“**独占所有内存/CPU**”的假象
    2. 内核中同样包含内核栈和内核代码、数据

如何表示进程：进程控制块(PCB)

> 每个进程对应于一个***元数据***，称为***进程控制块(PCB)***
>
> Q：进程控制块存储在内核态，这是为什么？
>
> A：因为需要PCB为用户态的各个进程提供“独占所有内存/CPU”的假象，需要由内核来控制

```
1 // ChCore 中的 PCB——process
2 struct process {
3     // 上下文
4     struct process_ctx *process_ctx;
5     // 虚拟地址空间
6     struct vmspace *vmspace;
7 };
```

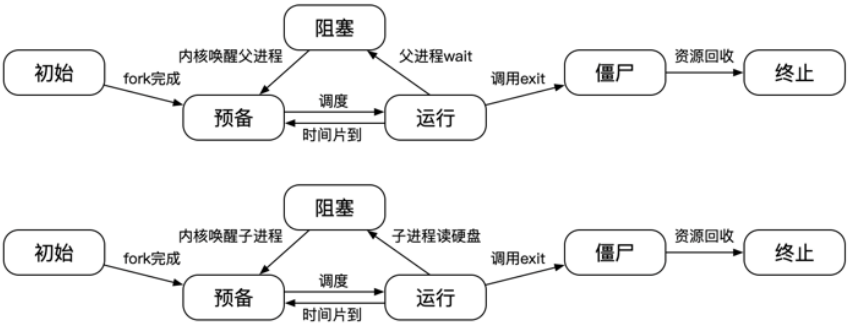
> 进程控制块中至少应该保存的信息：
>
> 1. 独立的虚拟地址空间（vmspace）
> 2. 独立的执行上下文(process_ctx)

简化模型：单一线程的进程

> 我们假设***每个进程都只有一个线程***
>
> 单一线程的进程中：线程管理/调度≈进程管理/调度

进程管理：管理进程的生命周期

- **进程自创建到终止可经历多个过程**
  - 称为进程**状态**
- **不同的系统调用和事件会影响进程的状态**



> 有一个idle进程，在“预备”中如果没有其它的进程等待，则会调用idle进程以节省运算资源

创建进程: 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**

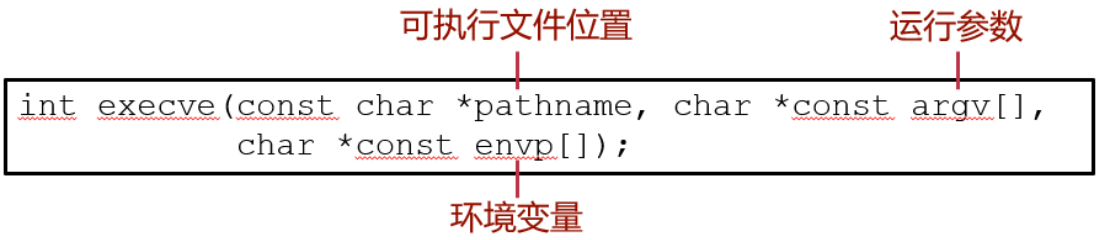
- fork后的两个进程均为独立进程

  > 1. 拥有不同的进程id
  > 2. 可以并行执行，互不干扰
  > 3. 父进程和子进程会共享部分数据结构

- fork的优点
  1. 接口非常的简介
  2. 将进程“创建”和“执行”(exec)解耦，提高了灵活度
  3. 刻画了进程之间内在的关系
- fork的缺点
  1. 完全拷贝过于粗暴(不如clone)
  2. 性能太差、可扩展性差(不如vfork和spawn)
  3. 不可组合性（比如fork()+pthread()）

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

> 早期的fork实现将父进程直接拷贝一份（性能差，无用功）
>
> 现在用***写时拷贝(Copy-On-Write)***，**只拷贝内存映射，不拷贝实际的内存**（性能较好，调用exec的情况下减少了无用的拷贝）

线程

为什么需要线程?

1. 创建进程的开销比较大

   > 数据、代码、堆、栈等

2. 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

3. 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***
  1. 静态部分由**进程**提供
  2. 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***
  1. 每个线程共享同一个地址空间
  2. 允许进程内并行（通过线程的方式）

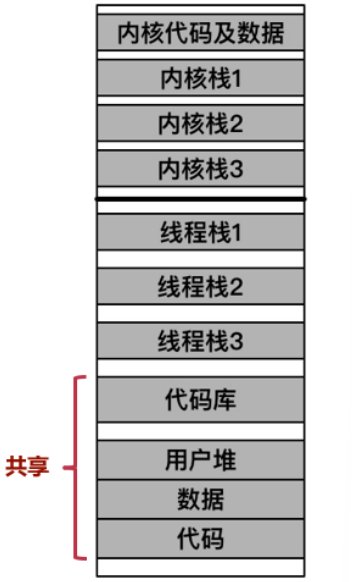
> “进程是资源分配的单位，线程是调度的单位”
>
> > 资源：内存、打开的文件等...
> >
> > 1. CPU分配计算时间是根据**进程**来的，而不是根据线程来分配
> > 2. 每个CPU都会有一个等待队列，这里面是**线程**

- ***一个进程的多线程可以在不同的处理器上同时执行***

1. 调度的基本单元/上下文切换的单位由进程变为了线程
2. 每个线程都有**状态**（此时进程的状态没有了意义）

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……



> 但注意，**堆(heap)**是由各个区域共享的

用户态线程与内核态线程

> 根据线程是否受内核管理划分

1. **内核态线程：内核可见，由内核管理**

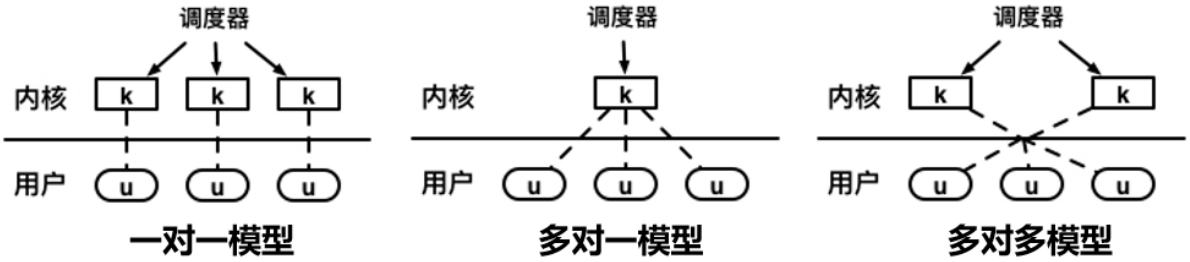
   > 由内核创建，线程相关信息存放在内核中
   >
   > (我们刚刚讲的典型线程就属于内核态线程)

2. **用户态线程（纤程）：内核不可见，不受内核直接管理**

   > 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



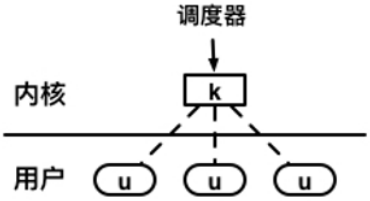
> 这里的“k”和“u”分别指的是内核栈和用户栈，**模型的对应关系就是内核栈与用户栈的对应关系**
>
> 例如一对一模型表达的意思就是一个用户栈对应于一个内核栈

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**

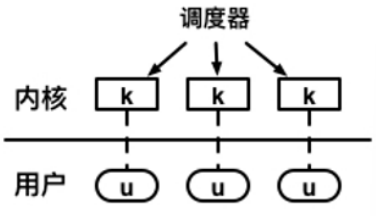
- **用于各种用户态线程库中**



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……

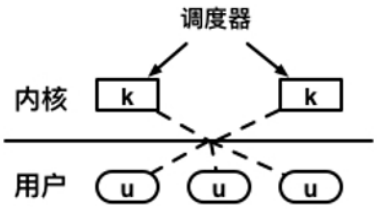


多对多模型（又叫Scheduler Activation）

- **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**



线程相关的数据结构：TCB

> "Thread Control Block"
>
> 一对一模型的TCB可以分为两部分

1. 内核态：与PCB类似

   > Linux中进程与线程中使用的是同一种数据结构(task_struct)
   >
   > 在上下文切换到的时候会用到

2. 应用态：可以由线程库自定义

> Linux: pthread结构体
>
> Windows：TIB(Thread Information Block)
>
> (类似于内核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

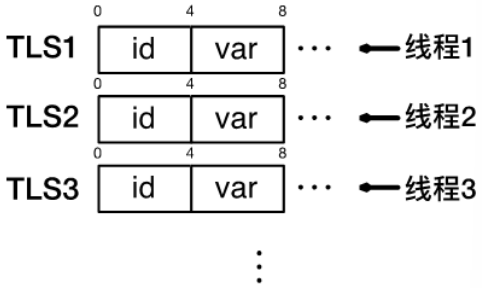
- **问题：对于全局变量，不同线程可能需要不同的拷贝**
  - 举例：用于标明系统调用错误的errno

- **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



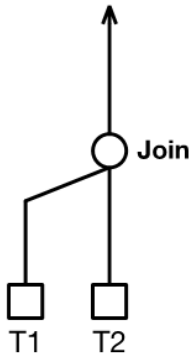
> Q: 一个有三个线程的进程调用了fork之后，新的进程有几个线程?
>
> A：只有一个，虽然三个线程的内核栈和线程栈都在新的process里面，但是另外两个不会被用到，新的PC指向的是调用线程中`fork`的下一行，用的是调用线程的栈

线程的基本操作

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”



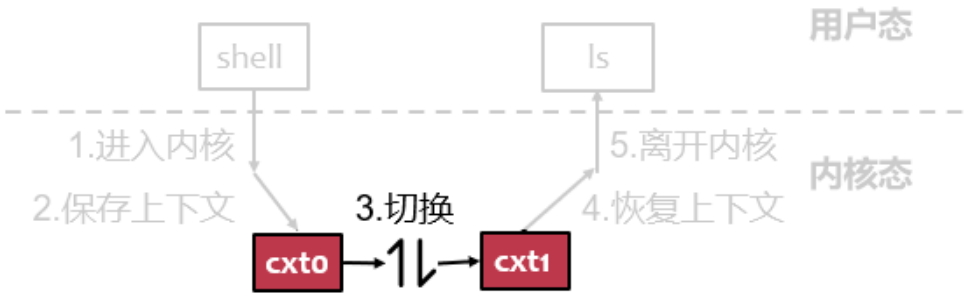
线程的基本操作：以*pthreads*为例

- **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

上下文切换

- 进程怎样切换到内核中执行？如何切换回用户态？
- 如何对上下文进行保存和恢复？
- 如何实现关键的切换步骤？



进程上下文的组成(AArch64)

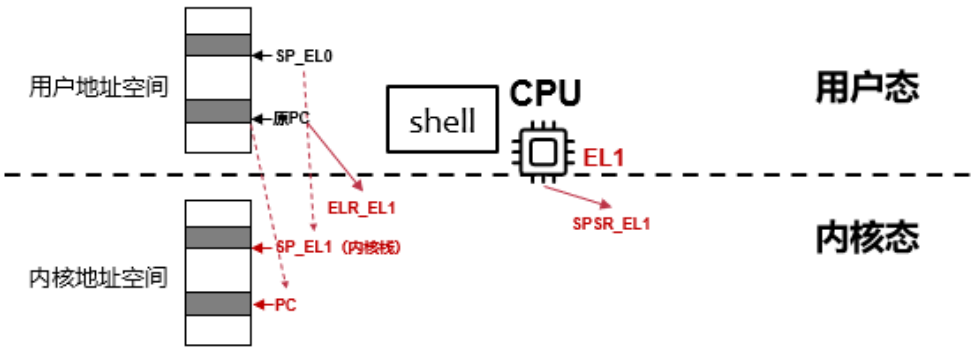
1. 常规寄存器：X0~X30
2. 程序计数器(PC)：保存在ELR_EL1
3. 栈指针：SP_EL0
4. CPU状态：保存在SPSR_EL1

> Q：为什么只需要保存寄存器信息，而不用保存内存信息？
>
> A：因为内存信息不会在context switch的过程中被改变，而寄存器信息会被覆盖

进程的内核态执行：切换到内核态

AArch64提供了硬件支持，使进程切换到内核态执行

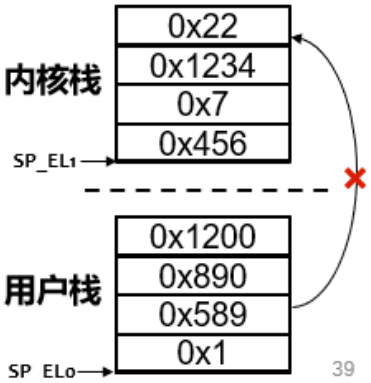
- 状态（PSTATE）写入SPSR_EL1
- 栈指针寄存器切换到SP_EL1
- PC移动到内核异常向量表中
- PC值写入ELR_EL1
- 运行状态切换到内核态EL1



进程的内核态执行：内核栈

- **为什么需要“又一个栈”（内核栈）？**
  - 进程在内核中依然执行代码，有读写临时数据的需求
  - 进程在用户态和内核态的数据应该相互隔离，增强安全性

- **AArch64实现：两个栈指针寄存器**
  - SP_EL1，SP_EL0
  - x86只有一个栈寄存器，需要保存恢复

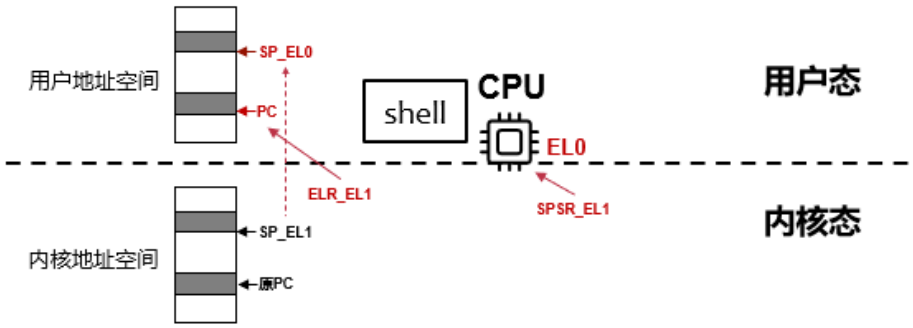


39

> **若不要内核栈，则存在安全漏洞**：若一个进程有两个线程同时执行，一个线程切换到内核态但仍然使用用户栈，另一个线程则可以通过修改用户栈中的内容来影响内核态线程的运行，可能造成安全上的一些问题

进程的内核态运行：返回用户态

- **进入内核态的“逆过程”，AArch64同样提供了硬件支持**
  - SPSR_EL1重设到CPU PSTATE
  - ELR_EL1重设到PC寄存器中
  - 栈指针寄存器切换到SP_EL0
  - 运行状态切换到用户态EL0
  - （为什么少了一步？）

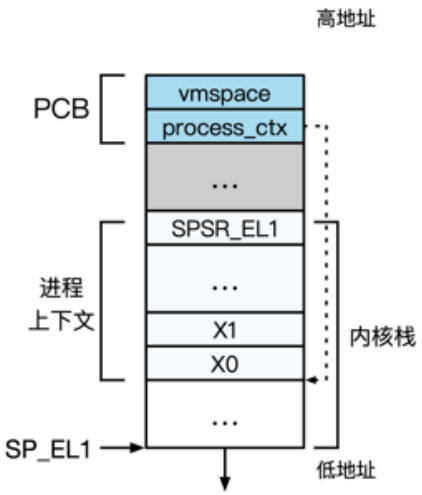


41

上下文保存

> 解决寄存器在切换后直接恢复导致的错误的问题
>
> ***保存上下文（寄存器）到内存，用于之后的恢复***

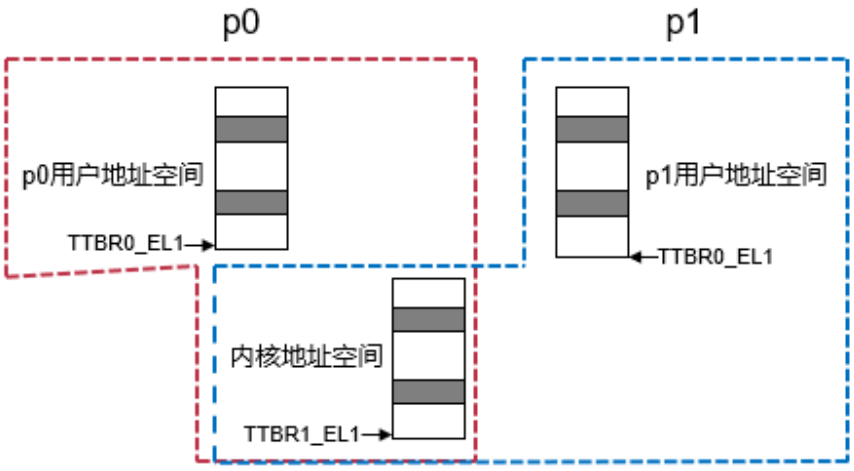
- 与进程相关的三种内核数据结构：PCB、上下文、内核栈
- PCB保存指向上下文的引用
- 上下文的位置固定在内核栈底部



> 在ChCore中：
>
> 1. 上下文保存：进入内核后调用exception_enter来完成->将各个寄存器逐一放入内核栈中
> 2. 上下文保存的逆过程：调用exception_exit来完成->从内核栈读出各个寄存器，并清空内核栈，最后调用`eret`

如何切换?

1. 如何切换到p1的地址空间?

> 由于用户地址独有，内核空间共享
>
> 
>
>
> 因此需要***获取p1的PCB，获取其vmspace，最后设置TTBR0_EL1***

2. 如何切换到p1的上下文

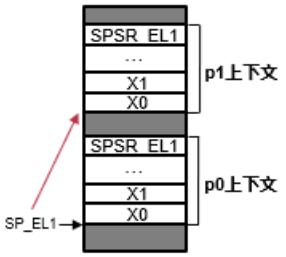
- **回顾：当p1保存上下文时，其内核栈应该是怎样的?**

SP_EL1指向上下文起始地址

- **p0/p1共享内核地址空间，因此直接切换内核栈指针即可**

```
u64 switch_context(void) {
    ...
    return pcb1->process_ctx;
}
```

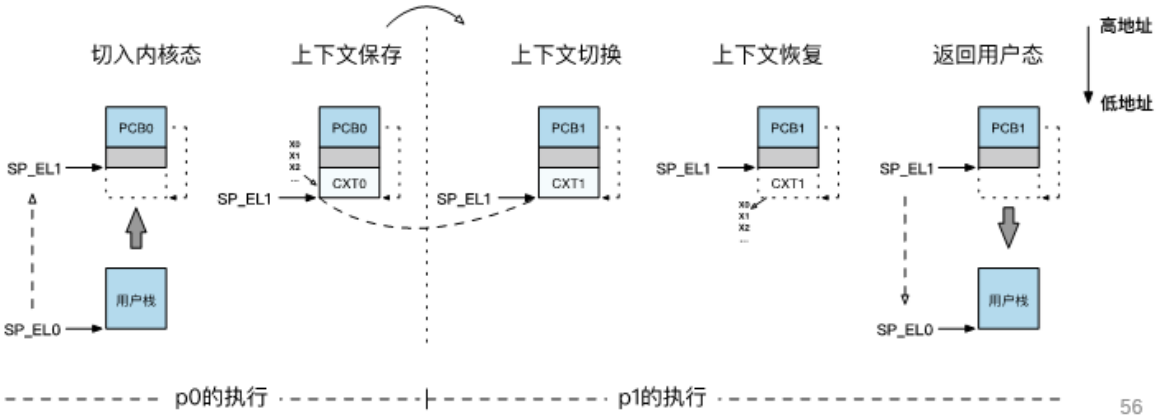
```
mov sp, x0
```



55

Summary

- **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”**



56

纤程

> 用户态线程

特性

- 比线程更加轻量级的运行时抽象
    1. 不单独对应内核线程
    2. 一个内核线程可以对应于多个纤程（多对一）
- 纤程的优点
    1. 不需要创建内核线程，开销小
    2. 上下文切换块（不需要进入内核）

3. 允许用户态自主调度，有助于做出更优的调度决策

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