上下文切换

CPU 上下文切换

大多数操作系统都是多任务，通常支持大于 CPU 数量的任务同时运行。实际上，这些任务并不是同时运行的，只是因为系统在很短的时间内，让各个任务分别在 CPU 运行，于是就造成同时运行的错觉。

任务是交给 CPU 运行的，那么在每个任务运行前，CPU 需要知道任务从哪里加载，又从哪里开始运行。

所以，操作系统需要事先帮 CPU 设置好 **CPU 寄存器和程序计数器。**

CPU 寄存器和程序计数是 CPU 在运行任何任务前，所必须依赖的环境，这些环境就叫做 **CPU 上下文。**

既然知道了什么是 CPU 上下文，那理解 CPU 上下文切换就不难了。

CPU 上下文切换就是先把前一个任务的 CPU 上下文（CPU 寄存器和程序计数器）保存起来，然后加载新任务的上下文到这些寄存器和程序计数器，最后再跳转到程序计数器所指的新位置，运行新任务。

系统内核会存储保持下来的上下文信息，当此任务再次被分配给 CPU 运行时，CPU 会重新加载这些上下文，这样就能保证任务原来的状态不受影响，让任务看起来还是连续运行。

上面说到所谓的「任务」，主要包含进程、线程和中断。所以，可以根据任务的不同，把 CPU 上下文切换分成：**中断上下文切换、进程上下文切换、线程上下文切换。**

中断上下文切换

对单个 CPU 上如何运行多个顺序进程的错觉做更多的解释。与每一 I/O 类相关联的是一个称作 `中断向量(interrupt vector)` 的位置（靠近内存底部的固定区域）。它包含中断服务程序的入口地址。假设当一个磁盘中断发生时，用户进程 3 正在运行，则中断硬件将程序计数器、程序状态字、有时还有一个或多个寄存器压入堆栈，计算机随即跳转到中断向量所指示的地址。这就是硬件所做的事情。然后软件就随即接管一切剩余的工作。

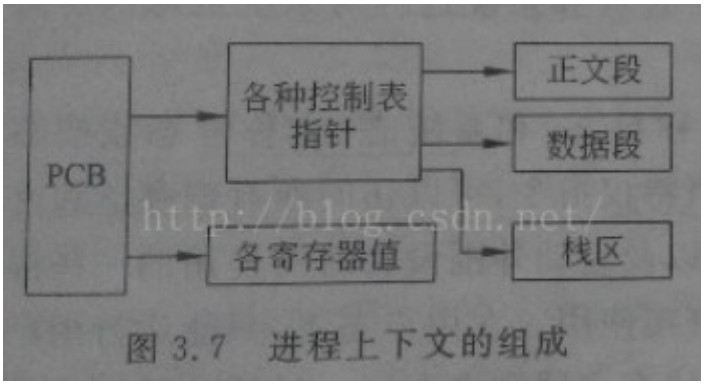
当中断结束后，操作系统会调用一个 C 程序来处理中断剩下的工作。在完成剩下的工作后，会使某些进程就绪，接着调用调度程序，决定随后运行哪个进程。然后将控制权转移给一段汇编语言代码，为当前的进程装入寄存器值以及内存映射并启动该进程运行，下面显示了中断处理和调度的过程。

1. 硬件压入堆栈程序计数器等
2. 硬件从中断向量装入新的程序计数器
3. 汇编语言过程保存寄存器的值
4. 汇编语言过程设置新的堆栈
5. C 中断服务器运行（典型的读和缓存写入）
6. 调度器决定下面哪个程序先运行
7. C 过程返回至汇编代码
8. 汇编语言过程开始运行新的当前进程

一个进程在执行过程中可能被中断数千次，但关键每次中断后，被中断的进程都返回到与中断发生前完全相同的状态。

进程上下文切换

各个进程之间是共享 CPU 资源的，在不同的时候进程之间需要切换，让不同的进程可以在 CPU 执行，那么这个**一个进程切换到另一个进程运行，称为进程的上下文切换。**进程是由内核管理和调度的，所以进程的切换只能发生在**内核态。**

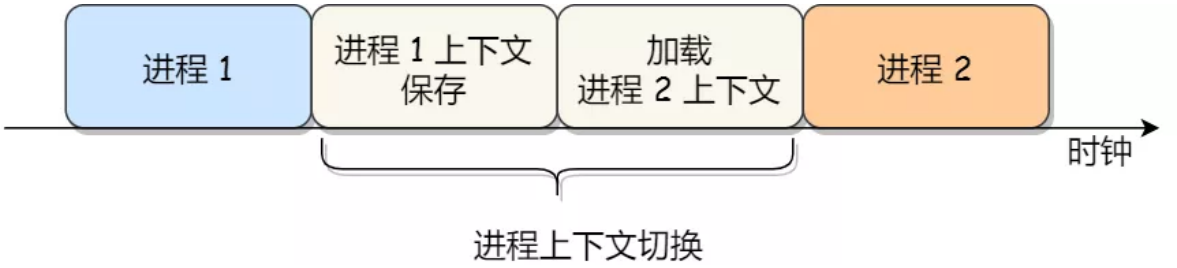


进程运行的环境称为进程上下文(context)，进程的上下文由进程控制块PCB(process control block)、正文段(text segment)、数据段(data segment)以及用户堆栈(stack)组成。其中:正文段存放该进程的可执行代码;数据段存放进程中静态产生的数据结构;PCB包括进程的编号、状态、优先级以及正文段和数据段中数据分布的大概情况。

进程上下文切换的内容

所以，**进程的上下文切换不仅包含了虚拟内存、栈、全局变量等用户空间的资源，还包括了内核堆栈、寄存器等内核空间的资源。**

通常，会把交换的信息保存在进程的 PCB，当要运行另外一个进程的时候，我们需要从这个进程的 PCB 取出上下文，然后恢复到 CPU 中，这使得这个进程可以继续执行，如下图所示：



进程上下文切换的场景

- 为了保证所有进程可以得到公平调度，CPU 时间被划分为一段段的时间片，这些时间片再被轮流分配给各个进程。这样，当某个进程的时间片耗尽了，就会被系统挂起，切换到其它正在等待 CPU 的进程运行；
- 进程在系统资源不足（比如内存不足）时，要等到资源满足后才可以运行，这个时候进程也会被挂起，并由系统调度其他进程运行；
- 当进程通过睡眠函数 sleep 这样的方法将自己主动挂起时，自然也会重新调度；
- 当有优先级更高的进程运行时，为了保证高优先级进程的运行，当前进程会被挂起，由高优先级进程来运行；
- 发生硬件中断时，CPU 上的进程会被中断挂起，转而执行内核中的中断服务程序；

进程上下文切换的巨大代价

另外需要说明的，上下文切换时保护现场和恢复现场都是有代价的，而且代价并不止于这里所描述的PCB保存和恢复。在一个进程正常运行过程中，CPU的各个寄存器和高速缓存中还保存了当前进程的很多数据和状态，在上下文切换时会丢失很多信息，在重新恢复该进程时，将刷新这些状态，不得不重新查询或计算。

所以，上下文切换频繁，必须考虑其代价问题，甚至有时候会将太多频繁的切换当作影响性能的关键指标。

正因为进程上下文切换的巨大代价，线程出现了。

线程的上下文切换

线程与进程最大的区别在于：**线程是调度的基本单位，而进程则是资源拥有的基本单位。**

所以，所谓操作系统的任务调度，实际上的调度对象是线程，而进程只是给线程提供了虚拟内存、全局变量等资源。

对于线程和进程，我们可以这么理解：

- 当进程只有一个线程时，可以认为进程就等于线程；
- 当进程拥有多个线程时，这些线程会共享相同的虚拟内存和全局变量等资源，这些资源在上下文切换时是不需要修改的；

另外，线程也有自己的私有数据，比如栈和寄存器等，这些在上下文切换时也是需要保存的。

> 线程上下文切换的是什么？

这还得看线程是不是属于同一个进程：

- 当两个线程不是属于同一个进程，则切换的过程就跟进程上下文切换一样；
- **当两个线程是属于同一个进程，因为虚拟内存是共享的，所以在切换时，虚拟内存这些资源就保持不动，只需要切换线程的私有数据、寄存器等不共享的数据**；

所以，线程的上下文切换相比进程，开销要小很多。

上下文内容	进 程	线 程
指向可执行文件的指针	√	
栈	√	√
内存(数据段和堆)	√	
状态	√	√
优先级	√	√
程序I/O的状态	√	
授予权限	√	
调度信息	√	
审计信息	√	
有关资源的信息● 文件描述符● 读/写指针	√	
有关事件和信号的信息	√	
寄存器组● 栈指针● 指令计数器● 诸如此类	√	√

区别与联系

进程切换涉及到虚拟地址空间的切换而线程切换则不会。因为每个进程都有自己的虚拟地址空间，而线程是共享所在进程的虚拟地址空间的，因此同一个进程中的线程进行线程切换时不涉及虚拟地址空间的转换。