线程

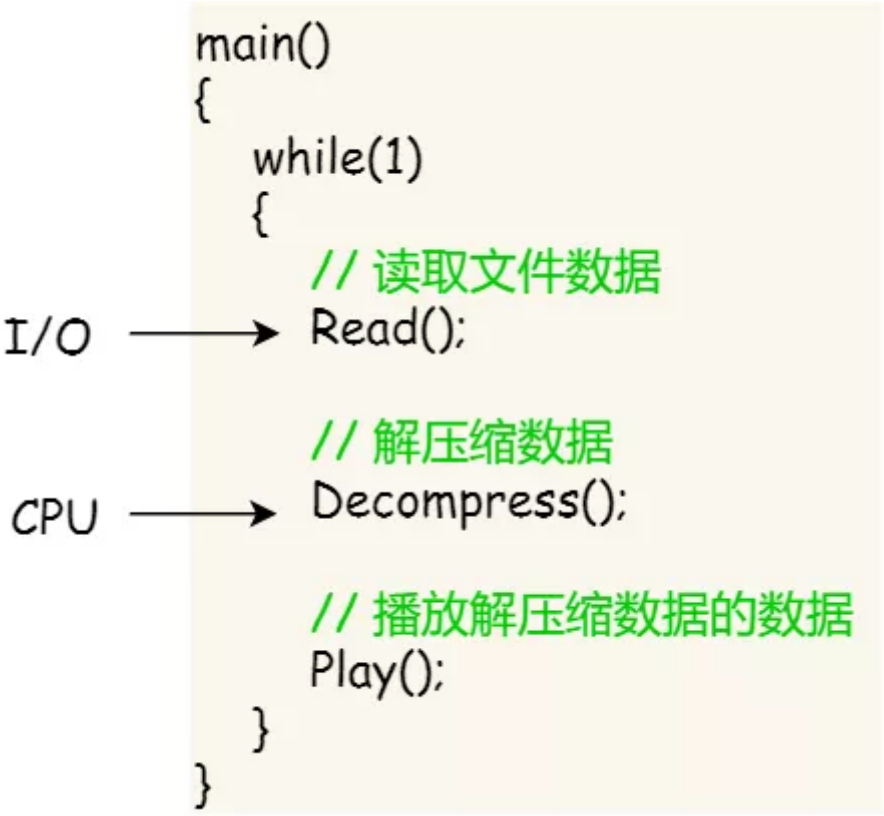
在早期的操作系统中都是以进程作为独立运行的基本单位，直到后面，计算机科学家们又提出了更小的能独立运行的基本单位，也就是**线程。**

为什么需要线程

我们举个例子，假设你要编写一个视频播放器软件，那么该软件功能的核心模块有三个：

- 从视频文件当中读取数据；
- 对读取的数据进行解压缩；
- 把解压缩后的视频数据播放出来；

单进程存在的问题



单进程实现方式

对于单进程的这种方式，存在以下问题：

- 播放出来的画面和声音会不连贯，因为当 CPU 能力不够强的时候，`Read` 的时候可能进程就等在这了，这样就会导致等半天才进行数据解压和播放；
- 各个函数之间不是并发执行，影响资源的使用效率；

多进程存在的问题

```
main()
{
    while(1)
    {
        // 读取文件数据
        Read();
    }
}
```

```
main()
{
    while(1)
    {
        // 解压缩数据
        Decompress();
    }
}
```

```
main()
{
    while(1)
    {
        // 播放解压缩数据的数据
        Play();
    }
}
```

多进程实现方式

对于多进程的这种方式，依然会存在问题：

- **进程之间如何通信，共享数据?**
- **维护进程的系统开销较大，如创建进程时，分配资源、建立 PCB；终止进程时，回收资源、撤销 PCB；进程切换时，保存当前进程的状态信息；**

线程如何解决问题

需要有一种新的实体，满足以下特性：

- 实体之间可以并发运行；
- 实体之间共享相同的地址空间；
- 实体更为轻量级，创建与撤销更快。

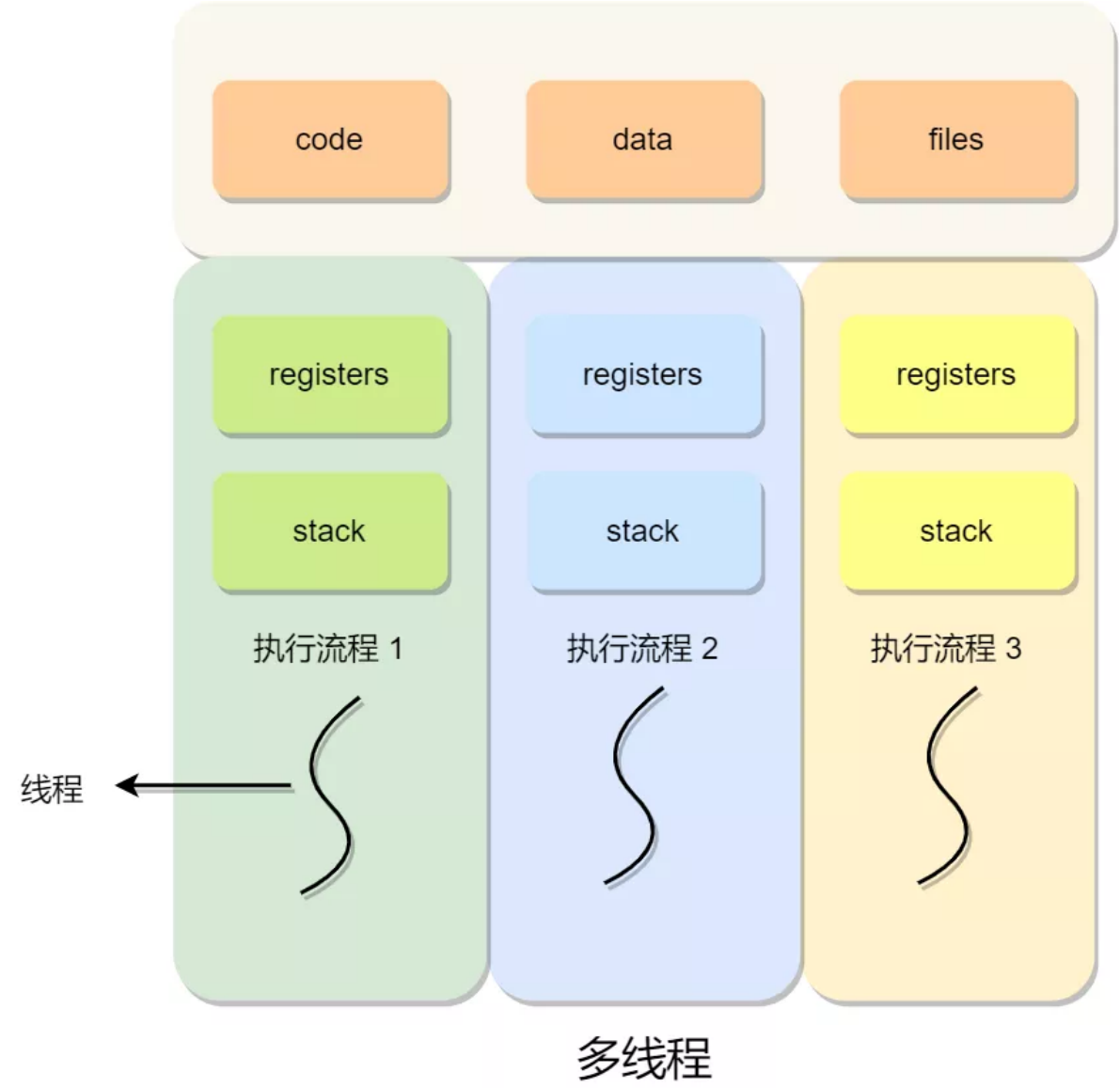
这个新的实体，就是**线程(Thread)**，线程之间可以并发运行且共享相同的地址空间。

什么是线程?

线程是进程当中的一条执行流程。

线程的结构

同一个进程内多个线程之间可以共享代码段、数据段、打开的文件等资源，但每个线程都有独立一套的寄存器和栈，这样可以确保线程的控制流是相对独立的。



线程的优缺点

线程的优点：

- 一个进程中可以同时存在多个线程；
- 各个线程之间可以并发执行；
- 各个线程之间可以共享地址空间和文件等资源；

线程的缺点：

- 当进程中的一个线程奔溃时，会导致其所属进程的所有线程奔溃。

举个例子，对于游戏的用户设计，则不应该使用多线程的方式，否则一个用户挂了，会影响其他同个进程的线程。

线程与进程的比较

线程与进程的差异：

- 进程是资源（包括内存、打开的文件等）分配的单位，线程是 CPU 调度的单位；
- 进程拥有一个完整的资源平台，而线程只独享必不可少的资源，如寄存器和栈；
- 线程同样具有就绪、阻塞、执行三种基本状态，同样具有状态之间的转换关系；
- 线程能减少并发执行的时间和空间开销；

线程相比进程能减少开销

- 线程的**创建**时间比进程快，因为进程在创建的过程中，还需要资源管理信息，比如内存管理信息、文件管理信息，而线程在创建的过程中，不会涉及这些资源管理信息，而是共享它们；
- 线程的**终止**时间比进程快，因为线程释放的资源相比进程少很多；
- 同一个进程内的线程**切换**比进程切换快，因为线程具有相同的地址空间（虚拟内存共享），这意味着同一个进程的线程都具有同一个页表，那么在切换的时候不需要切换页表。而对于进程之间的切换，切换的时候要把页表给切换掉，而页表的切换过程开销是比较大的；
- 由于同一进程的各线程间共享内存和文件资源，那么在线程之间**数据传递**的时候，就不需要经过内核了，这就使得线程之间的数据交互效率更高了；

所以，线程比进程不管是时间效率，还是空间效率都要高。

线程的实现

三种线程的实现方式

- **用户线程（*User Thread*）**：在用户空间实现的线程，不是由内核管理的线程，是由用户态的线程库来完成线程的管理；
- **内核线程（*Kernel Thread*）**：在内核中实现的线程，是由内核管理的线程；
- **轻量级进程（*LightWeight Process*）**：在内核中来支持用户线程；

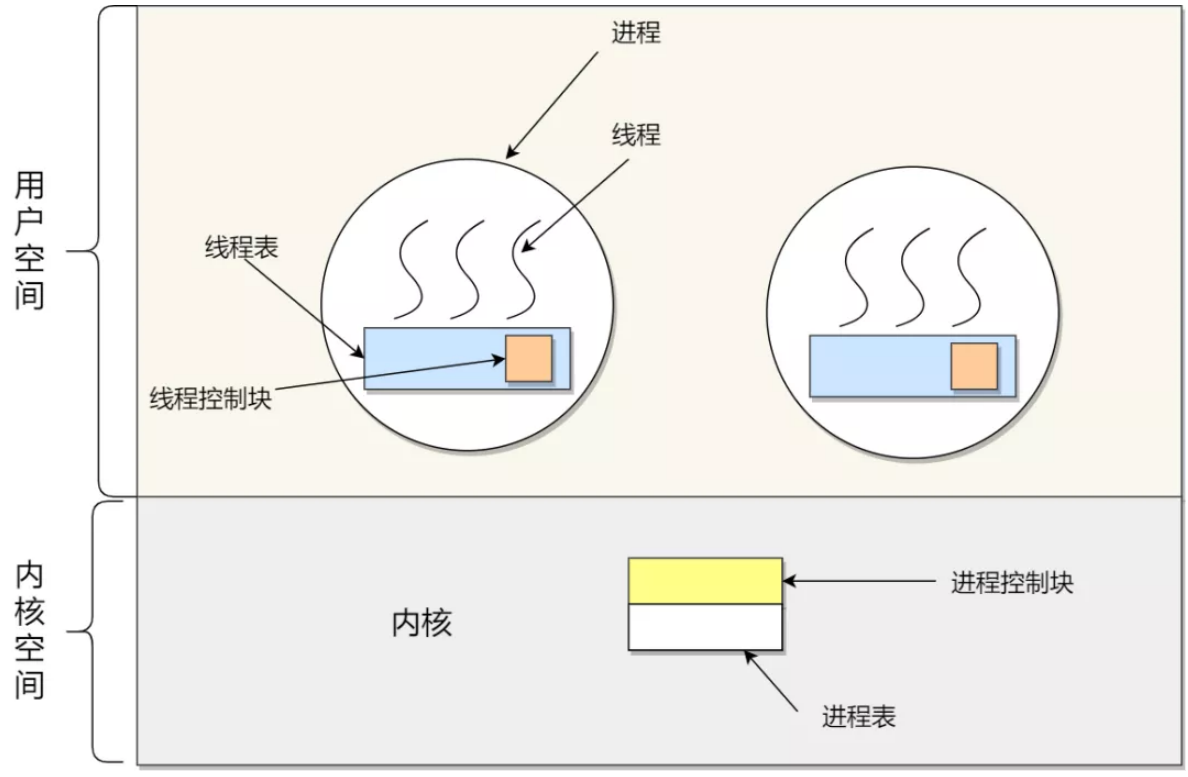
用户线程

用户线程的概念

用户线程是基于用户态的线程管理库来实现的，那么**线程控制块（Thread Control Block, TCB）** 也是在库里面来实现的，对于操作系统而言是看不到这个 TCB 的，它只能看到整个进程的 PCB。

所以，**用户线程的整个线程管理和调度，操作系统是不直接参与的，而是由用户级线程库函数来完成线程的管理，包括线程的创建、终止、同步和调度等。**

用户级线程的模型，也就类似前面提到的**多对一**的关系，即多个用户线程对应同一个内核线程，如下图所示：



用户线程的优点

- 每个进程都需要有它私有的线程控制块（TCB）列表，用来跟踪记录它各个线程状态信息（PC、栈指针、寄存器），TCB 由用户级线程库函数来维护，可用于不支持线程技术的操作系统；
- 用户线程的切换也是由线程库函数来完成的，无需用户态与内核态的切换，所以速度特别快；

用户线程的缺点

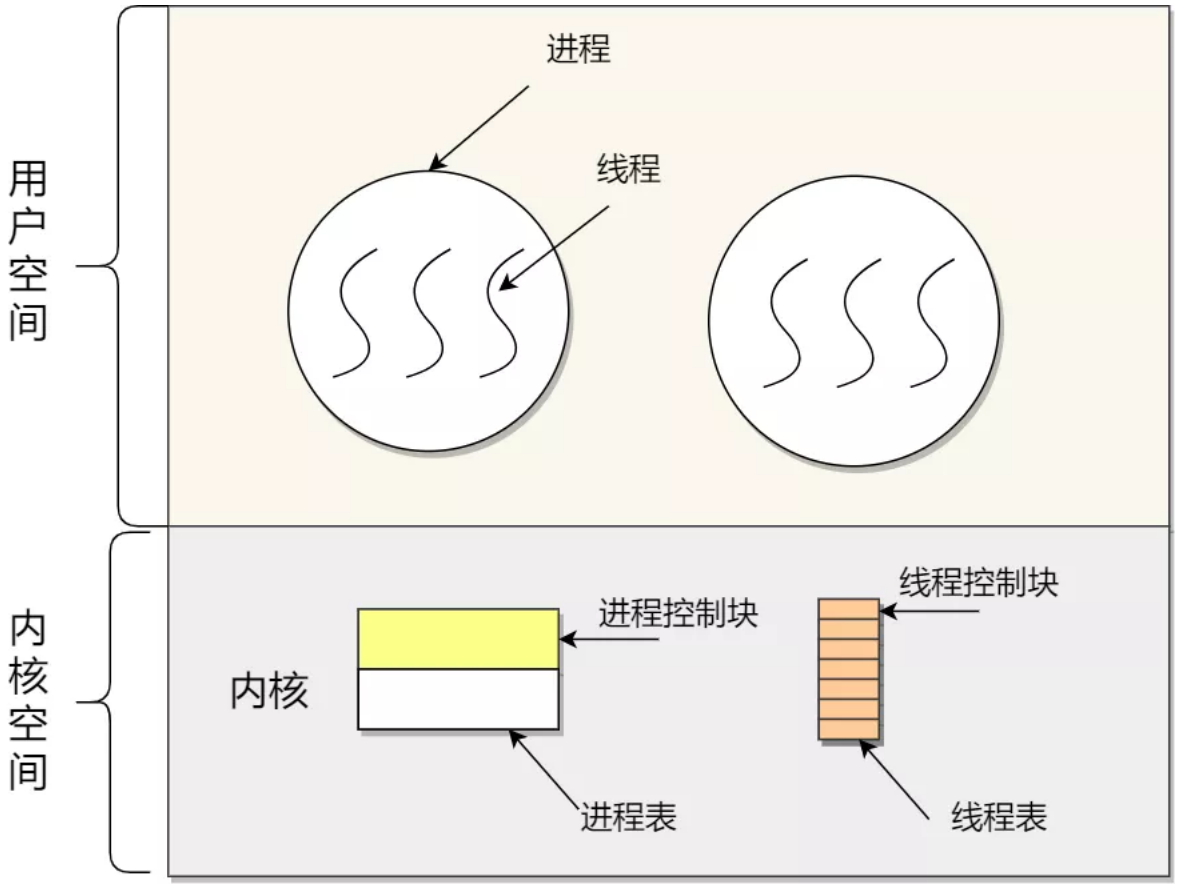
- 由于操作系统不参与线程的调度，如果一个线程发起了系统调用而阻塞，那进程所包含的用户线程都不能执行了。
- 当一个线程开始运行后，除非它主动地交出 CPU 的使用权，否则它所在的进程当中的其他线程无法运行，因为用户态的线程没法打断当前运行中的线程，它没有这个特权，只有操作系统才有，但是用户线程不是由操作系统管理的。
- 由于时间片分配给进程，故与其他进程比，在多线程执行时，每个线程得到的时间片较少，执行会比较慢；

内核线程

内核线程的概念

内核线程是由操作系统管理的，线程对应的 TCB 自然是放在操作系统里的，这样线程的创建、终止和管理都是由操作系统负责。

内核线程的模型，也就类似前面提到的**一对一**的关系，即一个用户线程对应一个内核线程，如下图所示：



内核线程的优点

- 在一个进程当中，如果某个内核线程发起系统调用而被阻塞，并不会影响其他内核线程的运行；
- 分配给线程，多线程的进程获得更多的 CPU 运行时间；

内核线程的缺点

- 在支持内核线程的操作系统中，由内核来维护进程和线程的上下文信息，如 PCB 和 TCB；
- 线程的创建、终止和切换都是通过系统调用的方式来进行，因此对于系统来说，系统开销比较大；

轻量级进程

轻量级进程（Light-weight process，LWP）是内核支持的用户线程，一个进程可有一个或多个 LWP，每个 LWP 是跟内核线程一对一映射的，也就是 LWP 都是由一个内核线程支持。

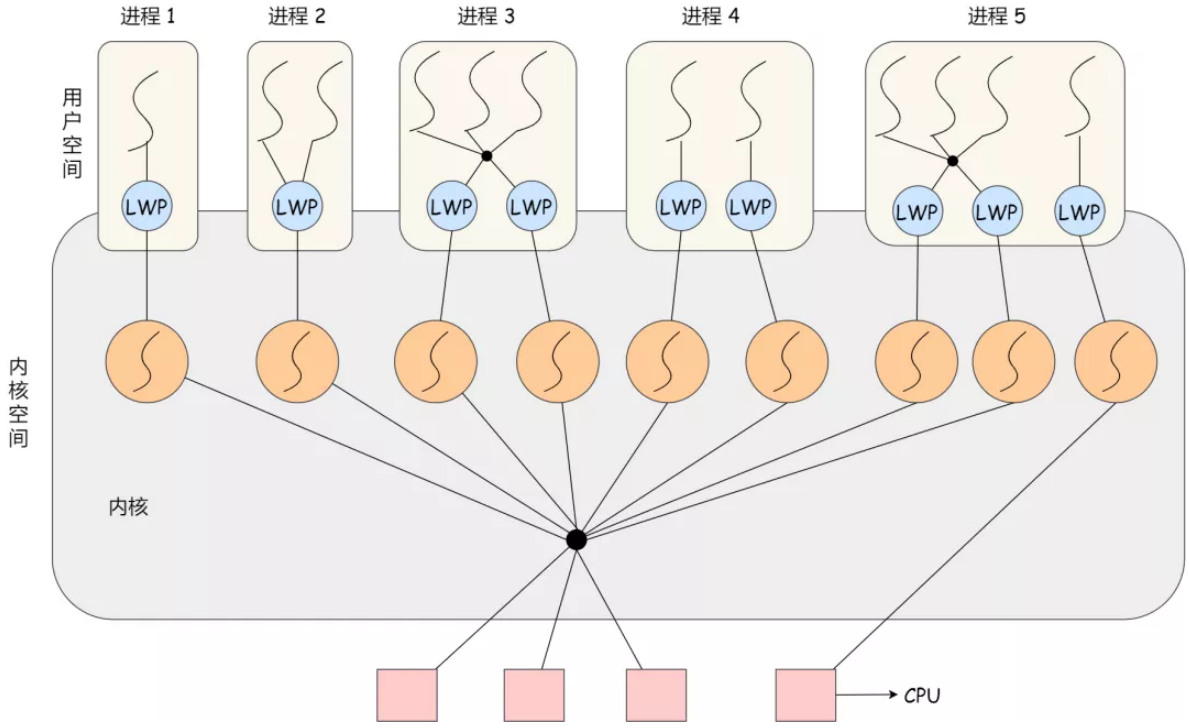
另外，LWP 只能由内核管理并像普通进程一样被调度，Linux 内核是支持 LWP 的典型例子。

在大多数系统中，**LWP与普通进程的区别也在于它只有一个最小的执行上下文和调度程序所需的统计信息。**一般来说，一个进程代表程序的一个实例，而 LWP 代表程序的执行线程，因为一个执行线程不像进程那样需要那么多状态信息，所以 LWP 也不带有这样的信息。

在 LWP 之上也是可以使用用户线程的，那么 LWP 与用户线程的对应关系就有三种：

- `1 : 1`，即一个 LWP 对应 一个用户线程；
- `N : 1`，即一个 LWP 对应多个用户线程；
- `N : N`，即多个 LMP 对应多个用户线程；

接下来针对上面这三种对应关系说明它们优缺点。先下图的 LWP 模型：



1 : 1 模式

一个线程对应到一个 LWP 再对应到一个内核线程，如上图的进程 4，属于此模型。

- 优点：实现并行，当一个 LWP 阻塞，不会影响其他 LWP；
- 缺点：每一个用户线程，就产生一个内核线程，创建线程的开销较大。

N : 1 模式

多个用户线程对应一个 LWP 再对应一个内核线程，如上图的进程 2，线程管理是在用户空间完成的，此模式中用户的线程对操作系统不可见。

- 优点：用户线程要开几个都没问题，且上下文切换发生用户空间，切换的效率较高；
- 缺点：一个用户线程如果阻塞了，则整个进程都将会阻塞，另外在多核 CPU 中，是没办法充分利用 CPU 的。

M : N 模式

根据前面的两个模型混搭一起，就形成 `M:N` 模型，该模型提供了两级控制，首先多个用户线程对应到多个 LWP，LWP 再一一对应到内核线程，如上图的进程 3。

- 优点：综合了前两种优点，大部分的线程上下文发生在用户空间，且多个线程又可以充分利用多核 CPU 的资源。

组合模式

如上图的进程 5，此进程结合 `1:1` 模型和 `M:N` 模型。开发人员可以针对不同的应用特点调节内核线程的数目来达到物理并行性和逻辑并行性的最佳方案。

Linux中的进程与线程

Linux使用LWP实现线程。

Linux 中进程和线程实际上都是用一个结构体 `task_struct` 来表示一个执行任务的实体。进程创建调用 `fork` 系统调用，而线程创建则是 `pthread_create` 方法，但是这两个方法最终都会调用到 `do_fork` 来做具体的创建操作 ，区别就在于传入的参数不同。

深究下去，你会发现 Linux 实现线程的方式简直太巧妙了，实际上根本没有**线程**，它创建的就是进程，只不过通过参数指定多个进程之间共享某些资源（如虚拟内存、页表、文件描述符等），函数调用栈、寄存器等线程私有数据则独立。

这样是不是非常符合理论书上的定义：同一进程内的多个线程共享该进程的资源，但线程并不拥有资源，只是使用他们。这也算符合 Unix 的哲学了— KISS（Keep It Simple, Stupid）。

但是在其它提供了专门线程支持的系统中，则会在进程控制块（PCB）中增加一个包含指向该进程所有线程的指针，然后再每个线程中再去包含自己独占的资源，比如 Windows 就是这样干的。

为什么选用轻量级进程

[我是一个进程]

我听说我的祖先们生活在专用计算机里， 一生只帮助人类做一件事情，比说微积分运算 了、人口统计了、生成密码、甚至通过织布机印花！

如果你想在这些专用"计算机"上干点别的事儿，例如安装个游戏玩玩， 那是绝对不可能的， 除非你把它拆掉， 然后建一个全新的机器。 而我这些祖先们勉强可以称为"程序"。

后来有个叫冯诺依曼的人， 非常了不起， 他提出了"存储程序"的思想， 并且把计算机分为五大部件：运算器、控制器、存储器、输入设备、输出设备。

各种各样不同功能的程序写好以后，和程序使用的数据一起存放在计算机的存储器中，即"存储程序"；然后，计算机按照存储的程序逐条取出指令加以分析，并执行指令所规定的操作。

这样一来，原来的专用计算机变成了通用的计算机，不管你是计算导弹弹道的，模拟核爆炸的，还是计算个人所得税的， 统统都可以在一台机器上运行， 我就是其中的一员： 专门计算员工的薪水。

进程的诞生

我所在的计算机是个批处理系统， 每次上机时， 我和其他程序都排好队， 一个接一个的进入内存运行。

每个月末是发薪日， 我都要运行一次，这样我每月都能见一次CPU阿甘， 这个沉默寡言，但是跑的非常快的家伙。

我知道内存看阿甘不顺眼，还告了它一状，说他一遇到IO操作的时候，就歇着喝茶 ，从来不管不问内存和硬盘的忙的要死的惨境。

其实我倒是觉得挺好， 这时候正好和阿甘海阔天空的聊天，他阅程序无数， 知道很多内部消息，每一个字节都清清楚楚， 和他聊天实在是爽。

又到了月末发薪水的时候， 我刚一进入内存，便看到这么一个公告：

> **公告**
>
> 为了创建和谐社会，促进效率和公平， 充分发挥每一个人的能力，经系统党委慎重研究决定：本系统自即日起，正式从“批处理系统”转为“多道程序系统”， 希望各部门通力配合，一起完成切换工作。
>
> 系统党委
>
> xxxx年xx月xx日

我正想着啥是多道程序系统， 阿甘便打电话给内存要我的指令开始运行了。

和之前一样，运行到了第13869123行，这是个IO指令， 我欢天喜地的准备和阿甘开聊了。

阿甘说： 哥们， 准备保存现场吧， 我要切换到另外一个程序来运行啦！

“啊？ 我这正运行着呢！ 咱们不喝茶了？

“喝啥茶啊，马上另外一个程序就来了!”

" 那我什么时候回来再见你？” 我问道。

“等这个IO指令完成，然后操作系统老大会再给你机会运行的。”

“那谁来记住我当前正在运行第13869123行？ 还有刚把两个数据从内存装载到了你的寄存器，就是那个EAX, EBX , 你一切换岂不都丢了？ ” 我有点着急。

阿甘说：“所以要暂时保存起来啊， 不仅仅是这些，还有你的那些函数在调用过程中形成的栈帧和栈顶，我这里用寄存器EBP和ESP维护着， 都得保存起来。”

“还有” 阿甘接着说，“你打开的文件句柄，你的程序段和数据段的地址， 你已经使用CPU的时间，等待CPU的时间。。。。。。 以及其他好多好多的东西，统统都要保存下来。”

我瞪大了眼睛： “这也太麻烦了吧， 原来我只需要关心我的指令和数据， 现在还得整这么多稀奇古怪的东西”

“没办法，这就叫做上下文切换， 把你的工作现场保存好，这样下一次运行的时候才能恢复啊。 对了，老大给你们统一起了一个新的名称： **进程**！刚才那些需要保存的东西叫做叫做**进程控制块**(Processing Control Block， **PCB**)，”

我想了想，这个名字还挺贴切的， 一个真正进行的程序！ 只是这个正在进行的程序随时可以被打断啊。

我只好保存好上下文，撤出CPU, 回到内存里歇着去了，与此同时另外一个程序开始占据CPU运行。

其实我这个程序，奥，不对，我这个进程被放到一个阻塞队列里，等到IO的数据来了以后，又被赶到了就绪队列中， 最后才有机会再次运行，再次见到CPU阿甘。

阿甘从我的PCB中取出各种保存的信息，恢复了运行时现场， 可是忙活了好一阵， 没办法，这就是程序切换必须要付出的代价。

我有点同情阿甘了， 从此以后，他很难再悠闲和和我们海阔天空，每时每刻都处于高速的奔跑中。

得益于阿甘的高速度， 虽然在同一时刻只有一个程序在运行， 但是有很多程序在短时间内不断的切换，在外界看来， 似乎多个程序在同时执行。

尤其是那些速度超慢的人类， 他们开着电脑一边听歌，一边上网，一边QQ，很是自在， 理所当然的认为这些程序就是同时在运行。 岂不知阿甘是让音乐播放器上运行几十毫秒， 然后打断，让浏览器进程运行几十毫秒，再打断，让QQ也运行几十毫秒，如此循环往复。

唉，阿甘真是能者多劳啊， 这个计算机系统也算是达到了我们党委的目标：兼顾了效率和公平。

有了进程就万事大吉了吗？ 人类的欲望是无止境的，很快就出现了新情况， 举个例子来说吧，我有一个兄弟，是个文字处理软件， 他和我不一样， 他有界面， 人类在用的时候能看到， 这实在是很幸福， 不像我总是在背后默默工作，几乎无人知晓。

这哥们有个智能的小功能，就是在人类编辑文档的时候能自动保存， 防止辛辛苦苦敲的文字由于断电什么的丢掉。

可是这个功能导致了人类的抱怨， 原因很简单，自动保存文字是和IO打交道，那硬盘有多慢你也知道，这个时候整个进程就被挂起了， 给人类的感觉就是： 程序死了，键盘和鼠标不响应了！ 无法继续输入文字， 但是过一会儿就好了。

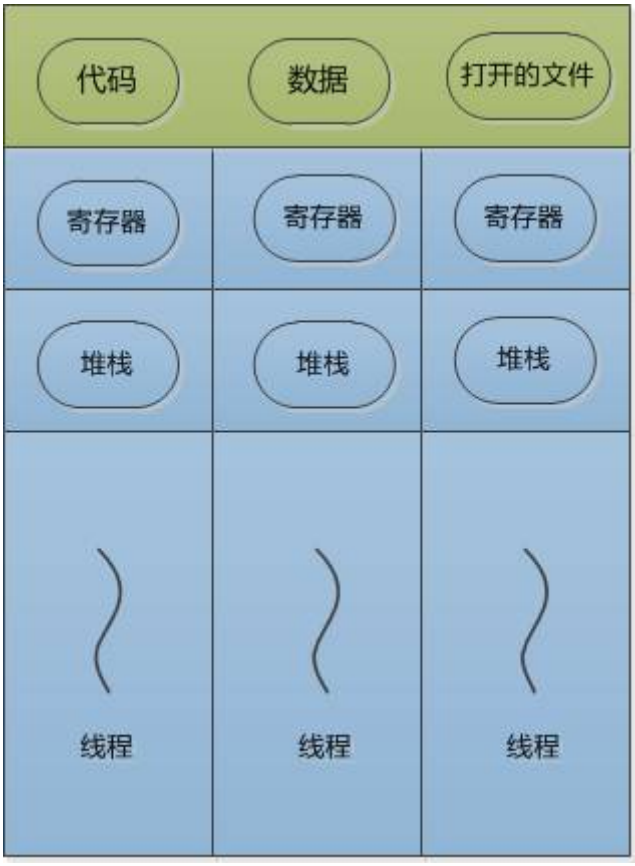
并且这种假死一会儿就会出现一次（每当自动保存的时候）， 让人不胜其烦。

系统党委研究了很久， 他们当然可以用两个进程来解决问题， 一个进程负责和用户交互， 另外一个进程负责自动保存， 但是，这两个进程之间完全是独立的，每个人都有自己的一亩三分地（地址空间），完全互不知晓， 进程之间通信的开销实在是太大， 他们没有办法高效的操作那同一份文档数据。

后来还是劳模阿甘想出了一招： 可以采用多进程的伟大思想啊！

把一个进程当成一个资源的容器， 让里边运行几个轻量级的进程， 就叫**线程**吧， 这些线程共享进程的所有资源， 例如地址空间，全局变量，文件资源等等。

但是每个线程也有自己独特的部分， 那就是要记住自己运行到哪一行指令了， 有自己的函数调用堆栈，自己的状态等等， 总而言之，就是为了能像切换进程那样切换线程。



拿我那个哥们的情况来说， 一个进程保存着文档的数据， 进程中有两个线程， 一个负责和用户交互，另外一个专门负责定时的自动保存， IO导致的阻塞就不会影响另外一个了。

注意，这两个线程都能访问进程的所有东西， 他们两个要小心，不要发起冲突才好 -- 这是人类程序员要做的事情了， 不归我们管。

争吵

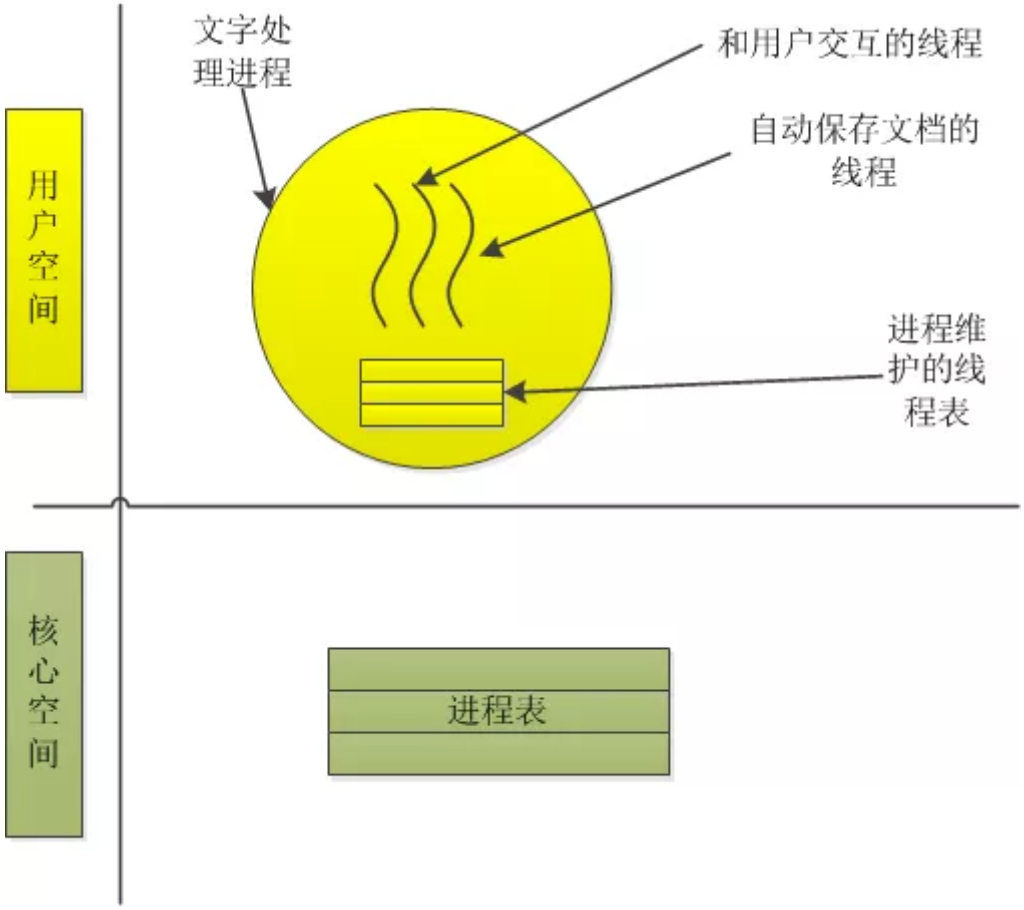
阿甘的建议被采纳了， 其实这几乎是唯一的解决问题方式了， 但是由谁来管理引起了激烈争吵。

系统党委有一波人坚持要在用户空间实现线程， 换通俗的话说就是让那些进程在自个儿内部去管理线程， 他们的理由也很充分：

你们自己实现了线程，可以自己定制自己的调度算法，多灵活啊；

所有的线程切换在进程内完成， 不用请求我们操作系统内核来处理，效率多高啊；

况且你们可以在那些内核不支持线程的操作系统中运行， 移植性多好啊。

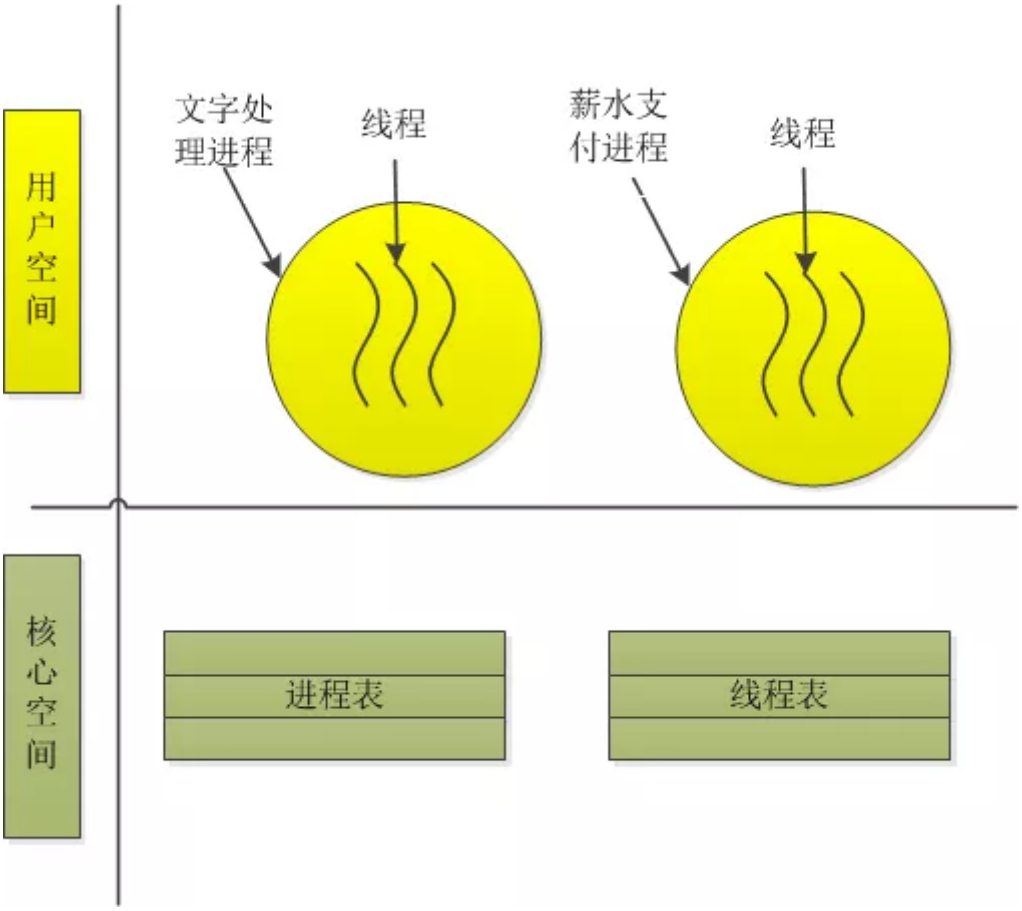


我们清楚的知道这是内核想做甩手掌柜， 因为他们选择性的忽略了一个致命的问题： 如果由我们实现线程，则操作系统内核还是认为我们只是一个进程而已，对里边的线程一无所知，对进程的调度还是以进程为最小单位。

一旦出现阻塞的系统调用，不仅仅阻塞那个线程，还会阻塞整个进程！

例如文字处理器那个进程， 如果负责定时保存的线程发起了IO调用， 内核会认为，这是由进程发起的，于是就把整个进程给挂起了， 虽然和用户交互的进程还是可以运行，也被强制的随着进程挂起来，不响应了， 这多么悲催啊， 又回到了老问题上去了。

所以我们坚决不能答应， 我们则一致的要求：在内核中实现线程！ 内核需要知道进程中线程的存在，内核需要维护线程表，并且负责调度！

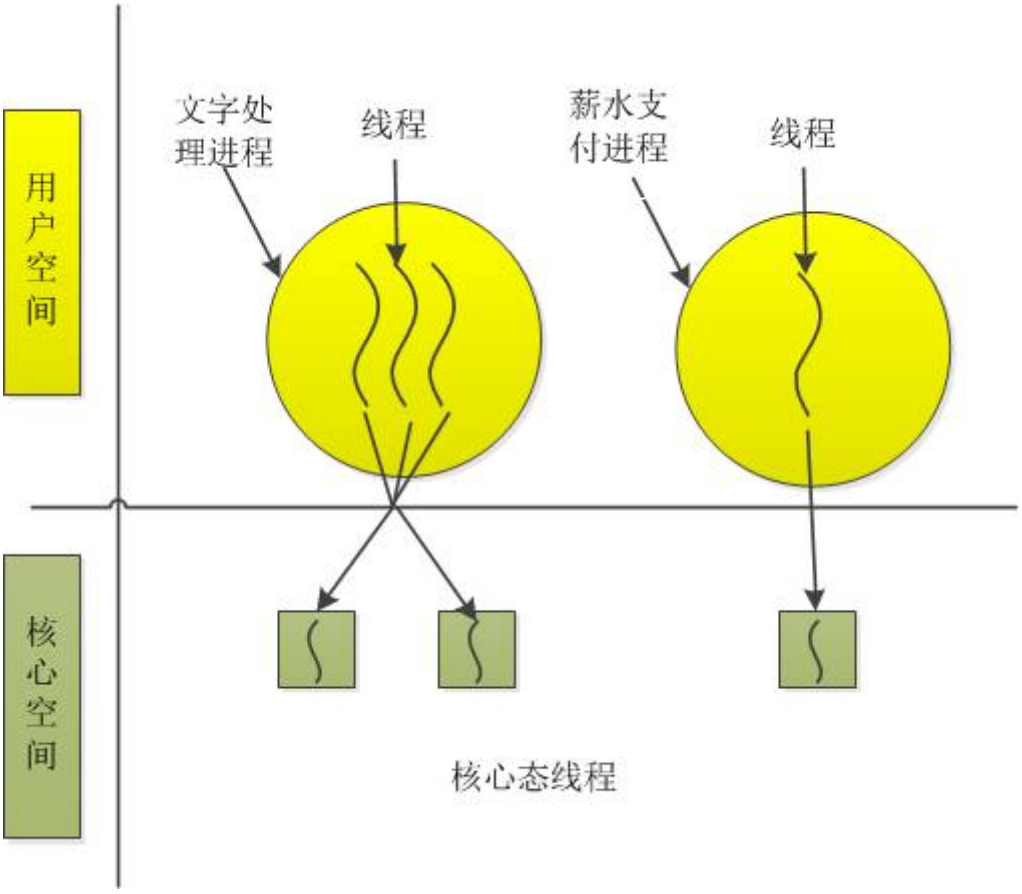


党委的人傲慢的说： 你们不嫌累吗， 每次创建一个线程都得通过我们内核， 多慢啊。

我们说：只有这样，一个线程的IO系统调用才不会阻塞我们整个进程啊， 你们完全可以选择同一个进程的另外一个线程去执行。

双发僵持不下，最后只好妥协， 那就是：混合着实现吧。

用户空间的进程可以创建线程（用户线程）， 内核也会创建线程（内核线程）， 用户线程映射到内核线程上。


用户空间
核心空间
文字处理进程
线程
薪水支付进程
线程
核心态线程