操作系统老大

又一个进程启动了，操作系统老大叹了一口气，毕竟自己的肩头又多了一份责任。

让人烦恼的是，新来的家伙们很无知，几乎就是一张白纸。有些老实本分的会按照自己的规矩来做事，有些刺头儿喜欢问这问那，时不时还想搞点非法的访问，想访问别的进程的地址空间，甚至想访问内核的代码和数据！ 这时候，我只有把他kill掉祭天，留下一个core dump的尸体让码农们去分析。

规矩很重要！

想到此处，老大又看了一眼自己的内核空间，这个机器只有可怜巴巴的4G内存，0-3G给各个进程共享使用，自己独占了从3G-4G的内存空间。

新启动的进程是一个Web服务器，自称小W，这是个喜欢问问题的家伙，他第一个问题就是： ”老大，你为啥不和群众打成一片，反而自己要独占空间呢？“

“这是为你们好？”

“为我们好？ ”

“计算机的硬件资源是有限的，硬盘、内存、网卡，键盘，鼠标，时钟...... 如果任由你们这些进程随意访问，大家你争我抢，岂不乱套？”

“再说了，那些底层的硬件、驱动操作是极其麻烦的，让你们每个进程都去写那些‘恶心’的代码，你们受得了吗？ “

”还有，如果某个恶意的家伙故意捣乱，那还了得？”

老大的三连问简直是振聋发聩， 小W立刻觉得气短了三分。

“所以你就不让我们直接访问了？”

“对啊，我就做了一个抽象层，你们必须通过这个抽象层来访问硬件资源。这个抽象层之下就是我的内核，是我的代码和数据，所以我必须得单独居住，不能和你们混在一起。”

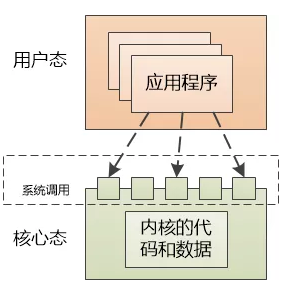
系统调用

“那我想访问一个硬盘上的文件，到底该怎么办？” 小W问道。

“非常简单，我的抽象层中有对外提供的接口，叫做系统调用，例如read、open、close等。 你可以open 一个文件，read它的内容，读完了close。”

“听起来好像是函数调用啊！”

“对，就是函数调用，但是和你内部的函数调用有本质的不同，这种系统调用会让你从用户态切换到内核态， 也就是到我的内核代码中来执行！”



小W懵懂地点点头，似乎明白了。

他应该没有明白，他也明白不了， 操作系统老大心里想，系统调用之复杂远远超过他的想象。

首先所有Linux的系统调用的参数都是通过寄存器而不是栈来传递的，按照惯例寄存器EAX保存了系统调用的编号（例如1表示exit这个系统调用，2表示fork,3表示read......），寄存器EBX,ECX,EDX,ESI,EDI可以包含最多6个任意的参数。

比如：write(1,"hello",5);

这就是个系统调用， 就是向stdout(控制台)输出一个字符串，在运行时，必须把寄存器给设置好：

EAX = 4 (4表示系统调用的编号)

EBX = 1 (1 表示stdout)

ECX = 那个字符串的地址

EDX = 字符串的长度

然后调用int 0x80 系统中断，这就进入了内核， 我会取出EAX， 从一个内核的表格中查到第4号对应的系统调用处理程序来执行。

对了，我还需要把CPU的特权等级从3置为0，表示内核态。

看看，我容易吗我！ 操作系统心里略微有点伤感。

read 和 write

这时候小W探出头来，兴奋地说：“hi ，老大，有客户要访问咱们硬盘的文件，我得读取一下，然后通过socket发出去。是不是需要系统调用了？”

“那是肯定的，访问文件系统必须得通过我，访问socket也得通过我，不用系统调用怎么可能？ 除去open ,close， 你需要两个关键的系统调用：”

// 从文件（用fd表示）中读取len长度的内容，放到buffer中

read(fd, buffer, len);

// 把buffer中长度为len 的内容写入到socket中(用sockfd表示)

write(sockfd, buffer, len);

（注： read和write 应该是sys\_read和sys\_write的“包裹”函数，我们这里简化，认为就是直接的函数调用。）

“好滴！” 小W做了一些准备工作，然后便开始read， 然后满心欢喜地等待数据的到来。

操作系统收到read调用，陷入内核，正式进入了内核态，然后毫不客气地暂停了小W的执行，让他进入了阻塞队列（假设小W只有一个线程）。

小W表示不满：“怎么不让我运行了？”

“读取文件太慢，你先歇会儿，数据来了会通知你的。”

老大使用DMA（Direct Memory Access）的方式把文件的数据从硬盘复制到了内核的缓冲区， 然后又复制到了用户的缓冲区，read调用完成，返回用户态 ，小W可以继续执行了。

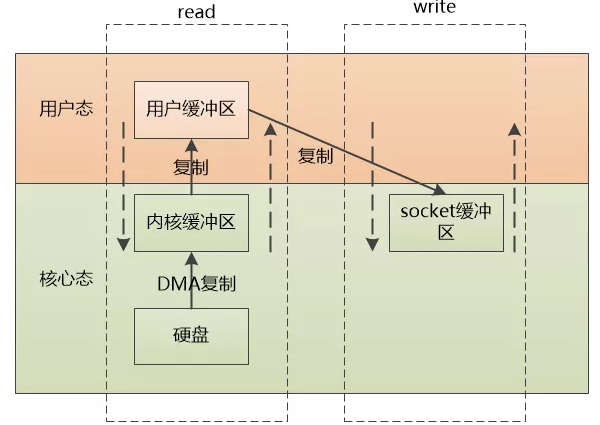
小W要通过socket发送数据，于是又发出了write调用，再次陷入内核，进入内核态。

老大把数据又从用户缓冲区复制到socket缓冲区， write调用返回，返回用户态。

小W问道：“这次这么快就返回了？数据发出去没有啊？”

老大说：“这就不用你操心了，网卡驱动会在合适的时候发送的，这是个异步的操作。”

小W画了一张图，试图理解整个过程，等他把图画完，不由得咂舌：“啧啧，这么两个简单的系统调用，代价竟然如此之高啊。”



(1) 需要进入内核态两次，返回两次。

(2) 数据居然发生了三次复制，硬盘-->内核缓冲区-->用户缓冲区-->socket缓冲区

如果说第一次从硬盘到内核缓冲区必不可少，后面的两次就太浪费了。

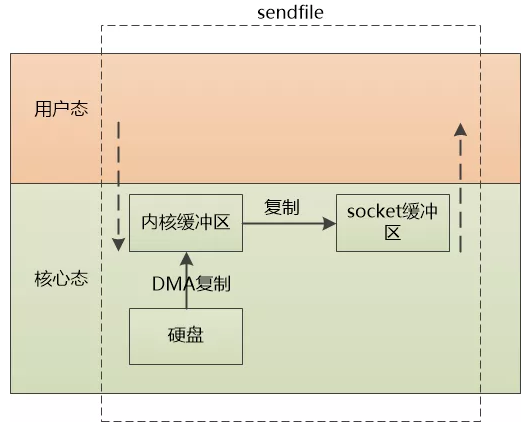
老大说：“你看到了吧，系统调用的开销很大啊，以后要少点调用啊。”

小W说：“我觉得你这个内核虽然保护了硬件，但是导致效率很低啊，能不能优化一下，省去用户态<-->核心态之间的数据复制？ 这太浪费了！”

sendfile

老大哈哈一笑， 说道： “我早就料到了这一层，我这里还有个系统调用，叫做sendfile，你可以试试啊，通过这个系统调用，可以直接把文件内容发给socket。 ”

sendfile(socket, file, len);



小W一看，不错啊，自己只需要调用sendfile，进入内核态一次就可以了，老大可以把数据从硬盘复制到内核缓冲区，然后直接复制到socket缓冲区， 完全不用自己介入，就用它了！

可是转念一想，这从内核缓冲区到socket缓冲区的复制有必要吗？ 那个网卡驱动不能直接从内核缓冲区读数据吗？

老大似乎看穿了小W的心思，说道：“我知道你在想啥，放心吧， 我早就做了优化了，不会把数据从内核缓冲区复制到socket缓冲区，相反，我只会把一些位置和数据长度等信息复制过去，很省事的。网卡驱动可以直接从内核缓冲区读去数据。”

小W放心了，开始使用这种sendfile的方式，果然，性能大为提升！

这其实就是所谓的zero copy技术， 从内核角度看，除了把文件从硬盘读出来之外，没有任何的额外copy。

zero copy技术减少了上下文的切换，避免了数据不断地在用户态和核心态搬运，不需要CPU参与数据的复制，提高了系统性能，在ngnix, apache等web 服务器中都引入了zero copy技术。