数组、链表和Cache污染

在hishscalability.com上看到一篇文章：Strategy: Stop Using Linked-Lists（这文章写的挺烂，只列结论没有具体分析），正好上周看到了一篇讲同样事情的文章为什么python标准库没有链表。两篇文章主题都是让大家写代码的时候少用链表，多用数组，主要原因是locality，局部性差的后果轻则cache miss，重则page fault。数组由于存储在连续空间里，其局部性显然是好过链表。

还记得大学操作系统课上老师说过：「局部性是编程里最重要的概念之一」，当时听的一知半解，现在算是有些感觉了。

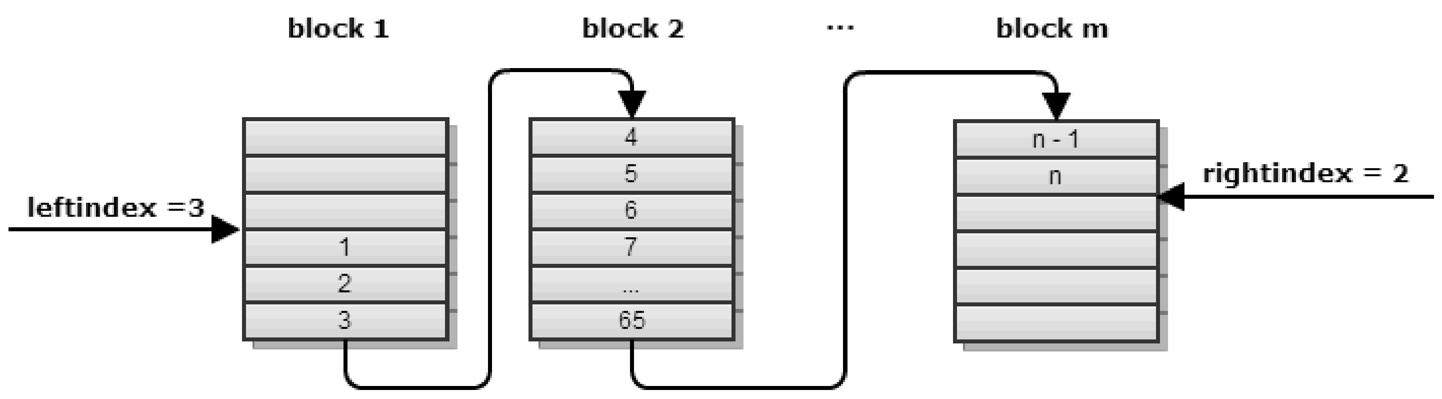
回到链表的问题。我们通常在什么情况下用链表呢？根据教科书上的大O复杂度，经常增删元素的情况下要用链表。这种场景下如果用数组替代链表，带来的overhead是数组的整体移动，按照流行的说法，内存拷贝很快，**但是内存拷贝有一个被人忽略副作用：被拷贝的内存进了CPU Cache（Cache污染）**，数组越大，污染的越厉害。现在CPU的L1 Cache大改是4~32KB，随便一个稍微大一点的数组就可以把整个L1 Cache全部污染。Cache污染使得程序的局部性被破坏。废话几句，内存相对于CPU来说，速度非常慢，CPU速度一直保持摩尔定律，而内存速度涨速甚缓。现在服务器CPU动辄2G 16核，而内存还不到2GHz，正因为如此才有了L1/L2/L3 Cache。

如何解决数组在插入删除时的Cache污染问题呢？

方法一

回到具体场景，链表的增删其实大多发生在头尾；对于大规模有序数据，非头尾数据频繁增删，树是一个更合适的数据结构。 对于头尾增删这种场景，第二篇文章中提到的python的deque的实现是个不错的方法（不过原作者理解错了），deque是一个以块为基本单位的双向链表，每个块可以存62个元素，leftindex和rightindex标记头尾两个块用到

哪了，这使得插入头尾的增删不需要任何元素移动。有兴趣的可以读一下源码。



PYTHON DEQUE

方法二

从另一个角度来考虑，能不能在内存复制的时候，避免cache污染呢？一个数据想进CPU寄存器，肯定会进Cache，这岂不是没招了。Google了一下，发现还是有办法的，不进CPU寄存器，还可以进FPU的寄存器（MMX寄存器），利用MMX和SSE指令完成内存拷贝。如下代码，不但完全不经过CPU Cache，而且使用了SIMD技术，每次可以拷贝16字节。

```
void memcpy(char * dst, char * src, unsigned size) {
    char * dst_end = dst + size;
    while (dst != dst_end) {
        __m128i res = _mm_stream_load_si128((__m128i *)src);
        *((__m128i *)dst) = res;
        src += 16;
        dst += 16;
    }
}
```

注1：引自Stack Overflow：How to use movntdqa to avoid cache pollution
注2：_mm_stream_load_si128是vs的私有函数，对SSE4的movntdqa的封装，MSDN链接

这个版本的memcpy又快又不污染cache，为什么没有成为c库或者kernel的实现呢？早在2000年Linus就和别人讨论过这个问题。Linus回复的大意如下：FPU的寄存器当然不是免费用的，每次使用都得保存现场恢复现场。因为FPU平时很少用，所以linux中有一个lasy FP switching的机制，加快task switch。如果memcpy使用FPU寄存器，这个机制基

本就失效了，从而使得task switching变慢，这是一个可怕的副作用。但是Linus也没把这个优化一棍子拍死：「如果你非要用，只在拷贝内存区比较大的情况下使用这个优化，小于几K就算了」。贴上Linus的回复，完整的讨论见：Page Zeroing Strategy。

> *Note that even in user space memcpy() using MMX registers is NOT necessarily a good idea at all.*
>
> *Why?*
>
> *It looks damn good in benchmarks. Especially for large memory areas that are not in the cache.*
>
> *But it tends to have horrible side-effects. Like the fact that when multiple processes (or threads) are running, it means that the FP state has to be switched all the time. Normally we can avoid this overhead, because most programs do not actually tend to use the FP unit very much, so with some simple lazy FP switching we can make thread and process switches much faster.*
>
> *Using the FPU or MMX for memcpy makes that go away completely. Suddenly you get slower task switching, and people will blame the kernel. Even though the _real_ bug is an optimization that looks very good on benchmarks, but does not necessarily actually win all that much in real life.*
>
> *Basically, you should almost never use the i387 for memcpy(), unless you know you can get it for free (ie you're already using the FPU). A i387 state save/restore is expensive. It's expensive even in user mode where you don't do it explicitly, but the kernel does it for you.*
>
> *The MMX stuff is similar. Only use it if you already know you're using the MXX unit. Because otherwise you _will_ slow the system down.*
>
> *NOTE! If you absolutely want to do it anyway, make sure that the size cutoff is large. It definitely is not worth a few FPU task switches to do small memcpy's. But for really large memcpy's you might consider it (ie if size is noticeably larger than a few kilobytes). Use regular integer stuff for smaller areas.*
>
> *And it's insidious. When benchmarking this thing, you usually (a) don't have any other programs running and (b) even if you do, they haven't been converted to using FPU memcpy yet anyway, so you'd see only half of the true cost anyway.*

最后，整个事说明了几个问题：

- 世界总是在变
- 没有免费的午餐

1条评论 5条新浪微博 最新 最早 最热

ThQzI

这个更刺激，准备好手纸哦 A 片。。 hTTp://uVU.cc/im7h

9月17日 回复 顶 转发

社交帐号登录: 微信 微博 QQ 人人 更多»

说点什么吧…

发布

Es gilt viele mauern abzubauen正在使用多说