Linux管道到底能有多快?

【CSDN 编者按】本文作者通过一个示例程序，演示了通过Linux管道读写数据的性能优化过程，使吞吐量从最初的 3.5GiB/s，提高到最终的65GiB/s。即便只是一个小例子，可它涉及的知识点却不少，包括零拷贝操作、环形缓冲区、分页与虚拟内存、同步开销等，尤其对Linux内核中的拼接、分页、虚拟内存地址映射等概念从源码级进行了分析。文章从概念到代码由浅入深、层层递进，虽然是围绕管道读写性能优化展开，但相信高性能应用程序或Linux内核的相关开发人员都会从中受益匪浅。

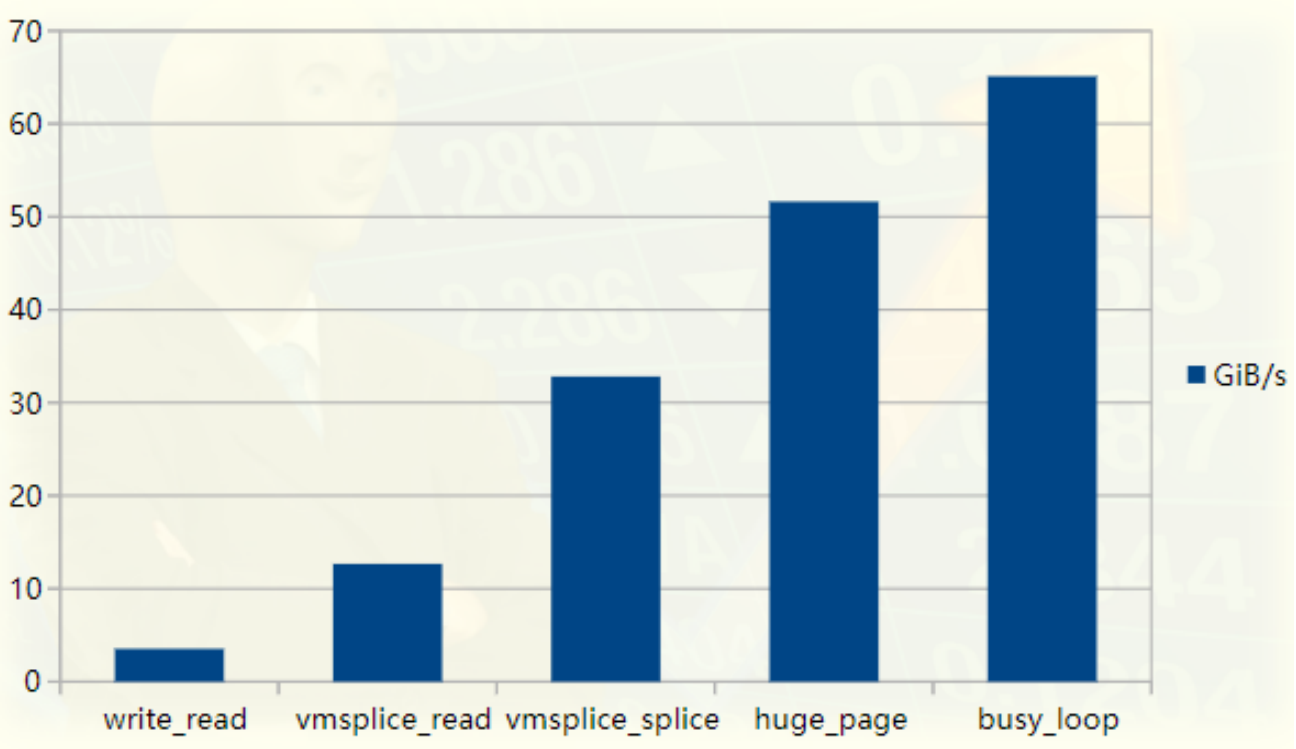
原文链接：https://mazzo.li/posts/fast-pipes.html

作者 | Francesco　　译者 |王雪迎

出品 | CSDN（ID：CSDNnews）

本文将对一个通过管道写入和读取数据的测试程序进行反复优化，以此研究 Unix 管道在 Linux 中的实现方式。

我们从一个吞吐量约为 3.5GiB/s 的简单程序开始，并逐步将其性能提升 20 倍。性能提升通过使用 Linux 的 perf tooling 分析程序加以确认，代码可从GitHub上获得（https://github.com/bitonic/pipes-speed-test）。



管道测试程序性能图

本文的灵感来自于阅读一个高度优化的 FizzBuzz 程序。在我的笔记本电脑上，该程序以大约 35GiB/s 的速度将输出推送到一个管道中。我们的第一个目标是达到这个速度，并会说明优化过程中每一步骤。之后还将增加一个 FizzBuzz 中不需要的额外性能改进措施，因为瓶颈实际上是计算输出，而不是 IO，至少在我的机器上是这样。

我们将按以下步骤进行：

1. 首先是一个管道基准测试的慢版本；
2. 说明管道内部如何实现，以及为什么从中读写会慢；

3. 说明如何利用vmsplice和splice系统调用，绕过一些（但不是全部！）缓慢环节；
4. 说明Linux分页，以及使用huge pages实现一个快速版本；
5. 用忙循环代替轮询以进行最后的优化；
6. 总结

第4步是 Linux 内核中最重要的部分，因此即使你熟悉本文中讨论的其他主题，也可能对它感兴趣。对于不熟悉相关主题的读者，我们假设你只了解 C 语言的基本知识。

挑战第一个慢版本

我们先按照 StackOverflow 的发帖规则，来测试传说中的 FizzBuzz 程序的性能：

```
% ./fizzbuzz | pv >/dev/null
 422GiB 0:00:16 [36.2GiB/s]
```

pv 指“pipe viewer”，是一种用于测量流经管道的数据吞吐量的简便工具。所示为 fizzbuzz 以 36GiB/s 的速率产生输出。

fizzbuzz 将输出写入与二级缓存一样大的块中，以在廉价访问内存和最小化 IO 开销之间取得良好平衡。

在我的机器上，二级缓存为 256KiB。本文中还是输出 256KiB 的块，但不做任何“计算”。我们本质上是想测量出程序写入具有合理缓冲区大小的管道的吞吐量上限。

fizzbuzz 使用 pv 测量速度，而我们的设置会略有不同：我们将在管道的两端执行程序，这样就可以完全控制从管道中推拉数据所涉及的代码。

该代码可从 in my pipes-speed-test rep 获得。write.cpp 实现写入，read.cpp 实现读取。write 一直重复写入相同的 256KiB，read 读取 10GiB 数据后终止，并以 GiB/s 为单位打印吞吐量。两个可执行程序都接受各种命令行选项以更改其行为。

从管道读取和写入的第一次测试将使用 write 和 read 系统调用，使用与 fizzbuzz 相同的缓冲区大小。下面所示为写入端代码：

```
int main() {
  size_t buf_size = 1 << 18; // 256KiB
  char* buf = (char*) malloc(buf_size);
  memset((void*)buf, 'X', buf_size); // output Xs
  while (true) {
    size_t remaining = buf_size;
    while (remaining > 0) {
      // Keep invoking `write` until we've written the entiret
      // of the buffer. Remember that write returns how muc
      // it could write into the destination -- in this case,
      // our pipe.
      ssize_t written = write(
        STDOUT_FILENO, buf + (buf_size - remaining), remai
      );
      remaining -= written;

    }
  }
}
```

为了简洁起见，这段及后面的代码段都省略了所有错误检查。memset 除了保证输出可被打印，还起到了另一个作用，我们将在后面讨论。

所有的工作都是通过 write 调用完成的，其余部分是确保整个缓冲区都被写入。读取端非常类似，只是将数据读取到 buf 中，并在读取足够的数据时终止。

构建后，资料库中的代码可以按如下方式运行：

```
% ./write | ./read
3.7GiB/s, 256KiB buffer, 40960 iterations (10GiB piped)
```

我们写入相同的 256KiB 缓冲区，其中填充了 40960 次“X”，并测量吞吐量。令人烦恼的是，速度比 fizzbuzz 慢 10 倍！我们只是将字节写入管道，而没做任何其他工作。事实证明，通过使用 write 和 read，我们无法获得比这快得多的速度。

write的问题

为了找出运行程序的时间花在了什么上，我们可以使用 perf：

```
% perf record -g sh -c './write | ./read'
3.2GiB/s, 256KiB buffer, 40960 iterations (10GiB piped)
[ perf record: Woken up 6 times to write data ]
[ perf record: Captured and wrote 2.851 MB perf.data (212
```

-g 选项指示 perf 记录调用图：这可以让我们从上到下查看时间花费在哪里。

我们可以使用 perf report 来查看花费时间的地方。下面是一个稍加编辑的片段，详细列出了 write 的时间花费所在：

```
% perf report -g --symbol-filter=write
-   48.05%    0.05%  write    libc-2.33.so      [.] __GI___libc_
  - 48.04% __GI___libc_write
     - 47.69% entry_SYSCALL_64_after_hwframe
        - do_syscall_64
           - 47.54% ksys_write
              - 47.40% vfs_write
                 - 47.23% new_sync_write
                    - pipe_write
                       + 24.08% copy_page_from_iter
                       + 11.76% __alloc_pages
                       + 4.32% schedule
                       + 2.98% __wake_up_common_lock
                         0.95% _raw_spin_lock_irq
                         0.74% alloc_pages
                         0.66% prepare_to_wait_event
```

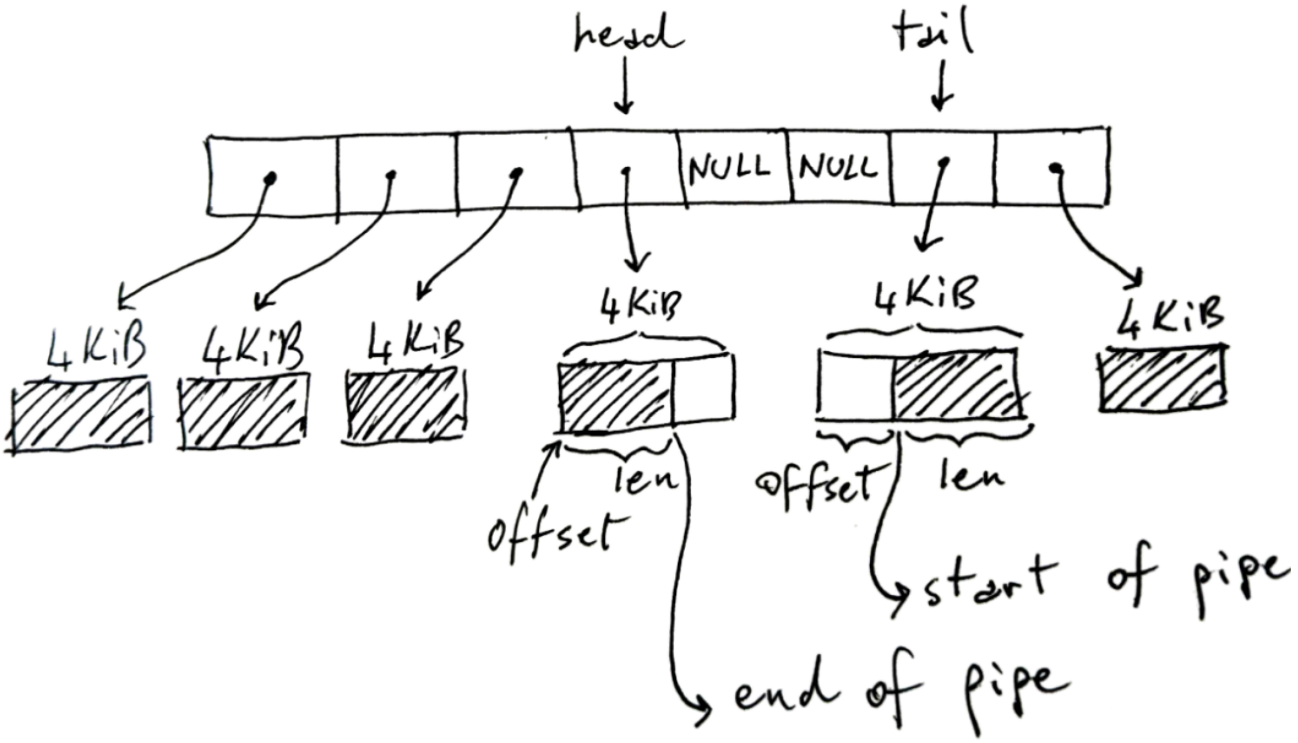
47% 的时间花在了 pipe_write 上，也就是我们在向管道写入时，write 所干的事情。这并不奇怪——我们花了大约一半的时间进行写入，另一半时间进行读取。

在 pipe_write中，3/4 的时间用于复制或分配页面（copy_page_from_iter和__alloc_page）。如果我们已经对内核和用户空间之间的通信是如何工作的有所了解，就会知道这有一定道理。不管怎样，要完全理解发生了什么，我们必须首先理解管道如何工作。

管道是由什么构成?

在 include/linux/pipe_fs_i.h 中可以找到定义管道的数据结构，fs/pipe.c 中有对其进行的操作。

Linux 管道是一个环形缓冲区，保存对数据写入和读取的页面的引用：



上图中的环形缓冲区有 8 个槽位，但可能或多或少，默认为 16 个。x86-64 架构中每个页面大小是 4KiB，但在其他架构中可能有所不同。这个管道总共可以容纳 32KiB 的数据。这是一个关键点：每个管道都有一个上限，即它在满之前可以容纳的数据总量。

图中的阴影部分表示当前管道数据，非阴影部分表示管道中的空余空间。

有点反直觉，head 存储管道的写入端。也就是说，写入程序将写入 head 指向的缓冲区，如果需要移动到下一个缓冲区，则相应地增加 head。在写缓冲区中，len 存储我们在其中写了多少。

相反，tail 存储管道的读取端：读取程序将从那里开始使用管道。offset 指示从何处开始读取。

注意，tail 可以出现在 head 之后，如图中所示，因为我们使用的是循环/环形缓冲区。还要注意，当我们没有完全填满管道时，一些槽位可能没有使用——中间的 NULL 单元。如果管道已满（页面中没有NULL和可用空间），write 将被阻塞。如果管道为空（全 NULL），则 read 将被阻塞。

下面是 pipe_fs_i.h 中 C 数据结构的节略版本：

```
struct pipe_inode_info {
  unsigned int head;
  unsigned int tail;
  struct pipe_buffer *bufs;
};

struct pipe_buffer {
  struct page *page;
  unsigned int offset, len;
};
```

这里我们省略了许多字段，也还没有解释 struct page 中存什么，但这是理解如何从管道进行读写的关键数据结构。

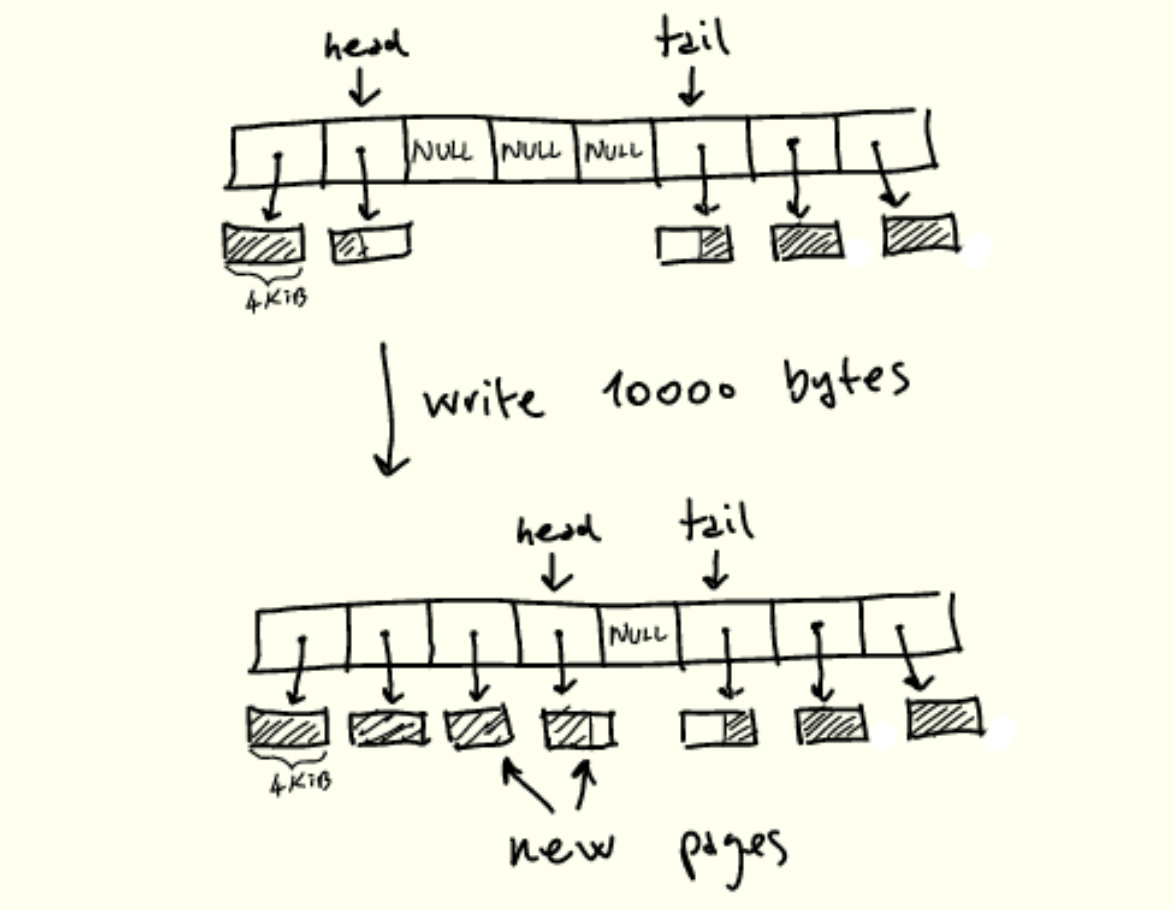
读写管道

现在让我们回到 pipe_write 的定义，尝试理解前面显示的 perf 输出。pipe_write 工作原理简要说明如下：

1.如果管道已满，等待空间并重新启动；

2.如果 head 当前指向的缓冲区有空间，首先填充该空间；

3.当有空闲槽位，还有剩余的字节要写时，分配新的页面并填充它们，更新head。



写入管道时的操作

上述操作被一个锁保护，pipe_write 根据需要获取和释放该锁。

pipe_read 是 pipe_ write 的镜像，不同之处在于消费页面，完全读取页面后将其释放，并更新 tail。

因此，当前的处理过程形成了一个令人非常不快的状况：

- 每个页面复制两次，一次从用户内存复制到内核，另一次从内核复制到用户内存；
- 一次复制一个 4KiB 的页面，期间还与诸如读写之间的同步、页面分配与释放等其他操作交织在一起；
- 由于不断分配新页面，正在处理的内存可能不连续；
- 处理期间需要一直获取和释放管道锁。

在本机上，顺序 RAM 读取速度约为 16GiB/s：

```
% sysbench memory --memory-block-size=1G --memory-o
...
102400.00 MiB transferred (15921.22 MiB/sec)
```

考虑到上面列出的所有问题，与单线程顺序 RAM 速度相比，慢 4 倍便不足为怪。

调整缓冲区或管道大小以减少系统调用和同步开销，或者调整其他参数不会有多大帮助。幸运的是，有一种方法可以完全避免读写缓慢。

用拼接进行改进

这种将缓冲区从用户内存复制到内核再复制回去，是需要进行快速 IO 的人经常遇到的棘手问题。一种常见的解决方案是将内核操作从处理过程中剔除，直接执行 IO 操作。例如，我们可以直接与网卡交互，并绕过内核以低延迟联网。

通常当我们写入套接字、文件或本例的管道时，首先写入内核中的某个缓冲区，然后让内核完成其工作。在管道的情况下，管道就是内核中的

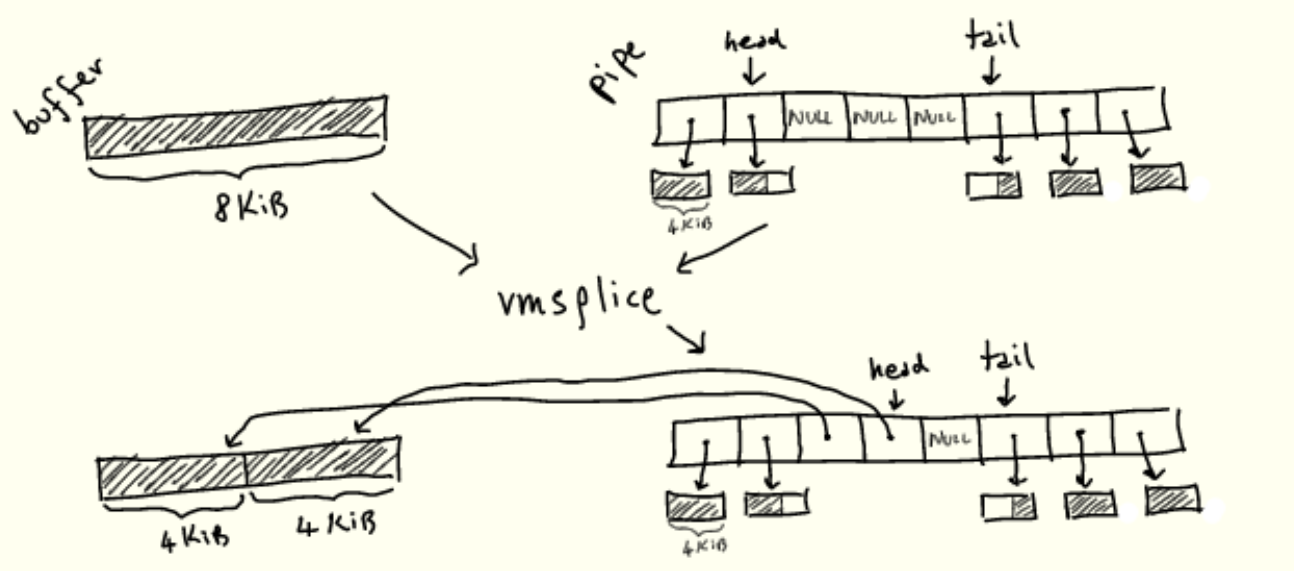
一系列缓冲区。如果我们关注性能，则所有这些复制都是不可取的。

幸好，当我们要在管道中移动数据时，Linux 包含系统调用以加快速度，而无需复制。具体而言：

- splice 将数据从管道移动到文件描述符，反之亦然；
- vmsplice 将数据从用户内存移动到管道中。

关键是，这两种操作都在不复制任何内容的情况下工作。

既然我们知道了管道的工作原理，就可以大概想象这两个操作是如何工作的：它们只是从某处“抓取”一个现有的缓冲区，然后将其放入管道环缓冲区，或者反过来，而不是在需要时分配新页面：



我们很快就会看到它是如何工作的。

Splicing 实现

我们用 vmsplice 替换 write。vmsplice 签名如下：

```
struct iovec {
```

```
  void  *iov_base; // Starting address
  size_t iov_len;  // Number of bytes
};

// Returns how much we've spliced into the pipe
ssize_t vmsplice(
  int fd, const struct iovec *iov, size_t nr_segs, unsigned in
);
```

fd是目标管道，struct iovec *iov 是将要移动到管道的缓冲区数组。注意，vmsplice 返回“拼接”到管道中的数量，可能不是完整数量，就像 write 返回写入的数量一样。别忘了管道受其在环形缓冲区中的槽位数量的限制，而 vmsplice 不受此限制。

使用 vmsplice 时还需要小心一点。用户内存是在不复制的情况下移动到管道中的，因此在重用拼接缓冲区之前，必须确保读取端使用它。

为此，fizzbuzz 使用双缓冲方案，其工作原理如下：

1. 将 256KiB 缓冲区一分为二；
2. 将管道大小设置为 128KiB，相当于将管道的环形缓冲区设置为具有 128KiB/4KiB=32 个槽位；
3. 在写入前半个缓冲区或使用 vmsplice 将其移动到管道中之间进行选择，并对另一半缓冲区执行相同操作。

管道大小设置为 128KiB，并且等待 vmsplice 完全输出一个 128KiB 缓冲区，这就保证了当完成一次 vmsplic 迭代时，我们已知前一个缓冲区已被完全读取——否则无法将新的 128KiB 缓冲区完全 vmsplice 到 128KiB 管道中。

现在，我们实际上还没有向缓冲区写入任何内容，但我们将保留双缓冲方案，因为任何实际写入内容的程序都需要类似的方案。

我们的写循环现在看起来像这样：

```
int main() {
  size_t buf_size = 1 << 18; // 256KiB
  char* buf = malloc(buf_size);
  memset((void*)buf, 'X', buf_size); // output Xs
  char* bufs[2] = { buf, buf + buf_size/2 };
  int buf_ix = 0;
  // Flip between the two buffers, splicing until we're dor
  while (true) {
    struct iovec bufvec = {
      .iov_base = bufs[buf_ix],
      .iov_len = buf_size/2
    };
    buf_ix = (buf_ix + 1) % 2;
    while (bufvec.iov_len > 0) {
      ssize_t ret = vmsplice(STDOUT_FILENO, &bufvec, 1,
      bufvec.iov_base = (void*) (((char*) bufvec.iov_base) +
      bufvec.iov_len -= ret;
    }
  }
}
```

以下是使用 vmsplice 而不是 write 写入的结果：

```
% ./write --write_with_vmsplice | ./read
12.7GiB/s, 256KiB buffer, 40960 iterations (10GiB piped)
```

这使我们所需的复制量减少了一半，并且把吞吐量提高了三倍多，达到 12.7GiB/s。将读取端更改为使用 splice 后，消除了所有复制，又获得了 2.5 倍的加速：

```
% ./write --write_with_vmsplice | ./read --read_with_splice
32.8GiB/s, 256KiB buffer, 40960 iterations (10GiB piped)
```

页面改进

接下来呢？让我们问 perf：

```
% perf record -g sh -c './write --write_with_vmsplice | ./rea
33.4GiB/s, 256KiB buffer, 40960 iterations (10GiB piped)
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.305 MB perf.data (241
% perf report --symbol-filter=vmsplice
-   49.59%     0.38%  write    libc-2.33.so       [.] vmsplice
  - 49.46% vmsplice
     - 45.17% entry_SYSCALL_64_after_hwframe
        - do_syscall_64
           - 44.30% __do_sys_vmsplice
              + 17.88% iov_iter_get_pages
              + 16.57% __mutex_lock.constprop.0
                3.89% add_to_pipe
                1.17% iov_iter_advance
                0.82% mutex_unlock
                0.75% pipe_lock
       2.01% __entry_text_start
       1.45% syscall_return_via_sysret
```

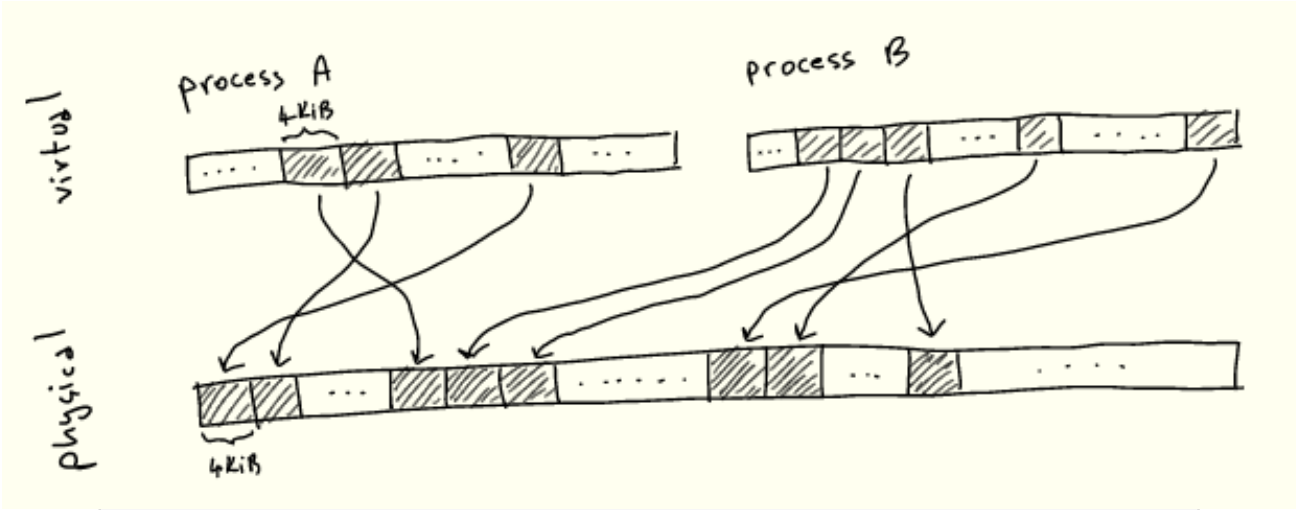
大部分时间消耗在锁定管道以进行写入（__mutex_lock.constprop.0）和将页面移动到管道中（iov_iter_get_pages）两个操作。关于锁定能改进的不多，但我们可以提高 iov_iter_get_pages 的性能。

顾名思义，iov_iter_get_pages 将我们提供给 vmsplice 的 struct iovecs 转换为 struct pages，以放入管道。为了理解这个函数的实际功能，以及如何加快它的速度，我们必须首先了解 CPU 和 Linux 如何组织页面。

快速了解分页

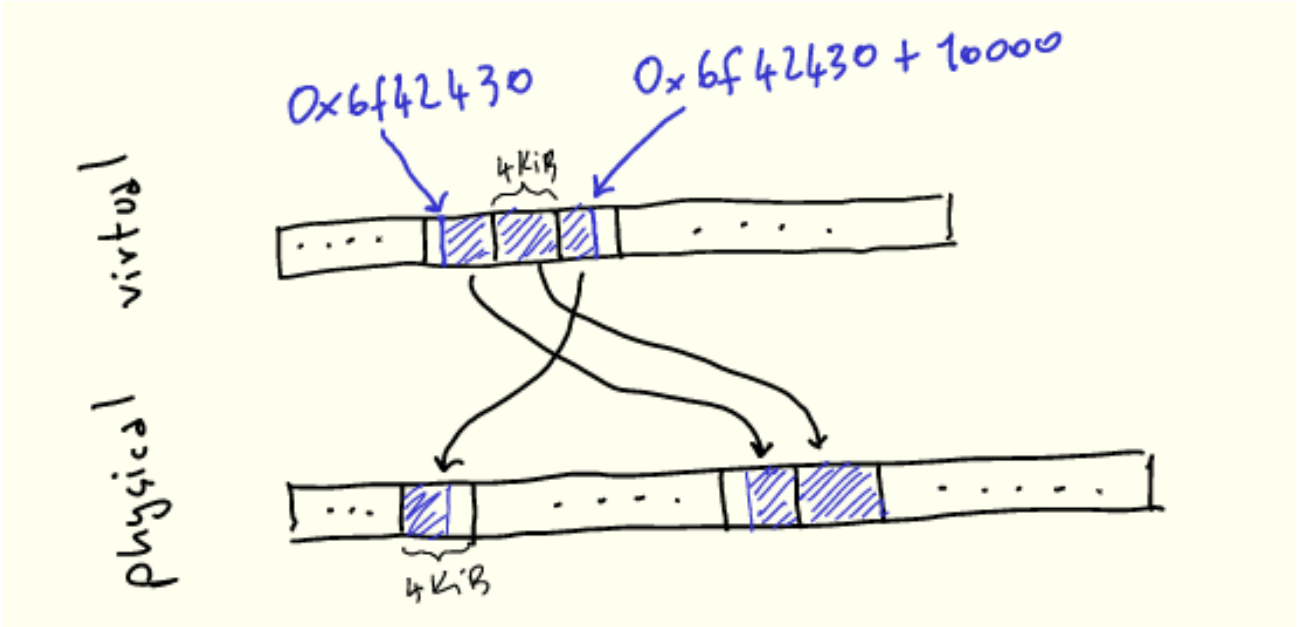
如你所知，进程并不直接引用 RAM 中的位置：而是分配虚拟内存地址，这些地址被解析为物理地址。这种抽象被称为虚拟内存，我们在这里不介绍它的各种优势——但最明显的是，它大大简化了运行多个进程对同一物理内存的竞争。

无论何种情况下，每当我们执行一个程序并从内存加载/存储到内存时，CPU 都需要将虚拟地址转换为物理地址。存储从每个虚拟地址到每个对应物理地址的映射是不现实的。因此，内存被分成大小一致的块，叫做页面，虚拟页面被映射到物理页面：




```
printf("%p\n", buf);         // 0x6f42430
```

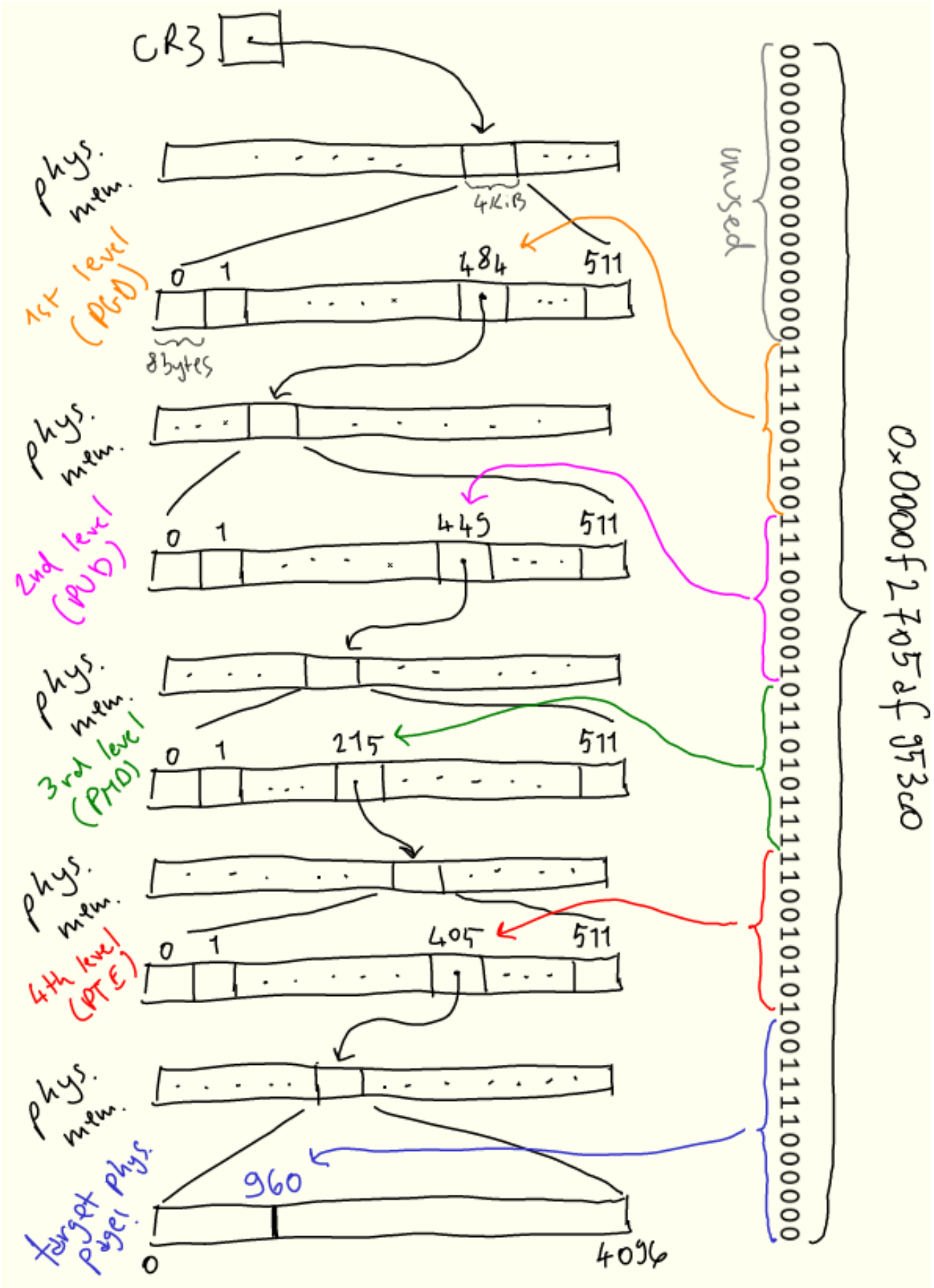
当我们使用它们时，我们的 10k 字节在虚拟内存中看起来是连续的，但将被映射到 3 个不必连续的物理页：



内核的任务之一是管理此映射，这体现在称为页表的数据结构中。CPU 指定页表结构（因为它需要理解页表），内核根据需要对其进行操作。在 x86-64 架构上，页表是一个 4 级 512 路的树，本身位于内存中。 该树的每个节点是（你猜对了！）4KiB 大小，节点内指向下一级的每个条目为 8 字节（4KiB/8bytes = 512）。这些条目包含下一个节点的地址以及其他元数据。

每个进程都有一个页表——换句话说，每个进程都保留了一个虚拟地址空间。当内核上下文切换到进程时，它将特定寄存器 CR3 设置为该树根的物理地址。然后，每当需要将虚拟地址转换为物理地址时，CPU 将该地址拆分成若干段，并使用它们遍历该树，以及计算物理地址。

为了减少这些概念的抽象性，以下是虚拟地址 0x0000f2705af953c0 如何解析为物理地址的直观描述：




```
struct pipe_inode_info {
  unsigned int head;
  unsigned int tail;
```

```
  struct pipe_buffer *bufs;
};

struct pipe_buffer {
  struct page *page;
  unsigned int offset, len;
};
```

然而，vmsplice 接受虚拟内存作为输入，而 struct page 直接引用物理内存。

因此我们需要将任意的虚拟内存块转换成一组 struct pages。这正是 iov_iter_get_pages 所做的，也是我们花费一半时间的地方：

```
ssize_t iov_iter_get_pages(
  struct iov_iter *i,  // input: a sized buffer in virtual memo
  struct page **pages, // output: the list of pages which bac
  size_t maxsize,      // maximum number of bytes to get
  unsigned maxpages,   // maximum number of pages to ge
  size_t *start        // offset into first page, if the input buffe
);
```

struct iov_iter 是一种 Linux 内核数据结构，表示遍历内存块的各种方式，包括 struct iovec。在我们的例子中，它将指向 128KiB 的缓冲区。vmsplice 使用 iov_iter_get_pages 将输入缓冲区转换为一组 struct pages，并保存它们。既然已经知道了分页的工作原理，你可以大概想象一下 iov_iter_get_pages 是如何工作的，下一节详细解释它。

我们已经快速了解了许多新概念，概括如下：

- 现代 CPU 使用虚拟内存进行处理；
- 内存按固定大小的页面进行组织；
- CPU 使用将虚拟页映射到物理页的页表，把虚拟地址转换为物理地址；
- 内核根据需要向页表添加和删除条目；
- 管道是由对物理页的引用构成的，因此 vmsplice 必须将一系列虚拟内存转换为物理页，并保存它们。

获取页的成本

在 iov_iter_get_pages 中所花费的时间，实际上完全花在另一个函数，get_user_page_fast 中：

```
% perf report -g --symbol-filter=iov_iter_get_pages
-   17.08%     0.17%  write    [kernel.kallsyms]  [k] iov_iter_g
   - 16.91% iov_iter_get_pages
      - 16.88% internal_get_user_pages_fast
           11.22% try_grab_compound_head
```

get_user_pages_fast 是 iov_iter_get_ pages 的简化版本：

```
int get_user_pages_fast(
  // virtual address, page aligned
  unsigned long start,
  // number of pages to retrieve
  int nr_pages,
  // flags, the meaning of which we won't get into
  unsigned int gup_flags,
  // output physical pages
  struct page **pages
```

```
)
```

这里的“user”（与“kernel”相对）指的是将虚拟页转换为对物理页的引用。

为了得到 struct pages，get_user_pages_fast 完全按照 CPU 操作，但在软件中：它遍历页表以收集所有物理页，将结果存储在 struct pages 里。我们的例子中是一个 128KiB 的缓冲区和 4KiB 的页，因此 nr_pages = 32。get_user_page_fast 需要遍历页表树，收集 32 个叶子，并将结果存储在 32 个 struct pages 中。

get_user_pages_fast 还需要确保物理页在调用方不再需要之前不会被重用。这是通过在内核中使用存储在 struct page 中的引用计数来实现的，该计数用于获知物理页在将来何时可以释放和重用。get_user_pages_fast 的调用者必须在某个时候使用 put_page 再次释放页面，以减少引用计数。

最后，get_user_pages_fast 根据虚拟地址是否已经在页表中而表现不同。这就是 _fast 后缀的来源：内核首先将尝试通过遍历页表来获取已经存在的页表条目和相应的 struct page，这成本相对较低，然后通过其他更高成本的方法返回生成 struct page。我们在开始时memset内存的事实，将确保永远不会在 get_user_pages_fast 的“慢”路径中结束，因为页表条目将在缓冲区充满“X”时创建。

注意，get_user_pages 函数族不仅对管道有用——实际上它在许多驱动程序中都是核心。一个典型的用法与我们提及的内核旁路有关：网卡驱动程序可能会使用它将某些用户内存区域转换为物理页，然后将物理

页位置传递给网卡，并让网卡直接与该内存区域交互，而无需内核参与。

大体积页面

到目前为止，我们所呈现的页大小始终相同——在 x86-64 架构上为 4KiB。但许多 CPU 架构，包括 x86-64，都包含更大的页面尺寸。x86-64 的情况下，我们不仅有 4KiB 的页（“标准”大小），还有 2MiB 甚至 1GiB 的页（“巨大”页）。在本文的剩余部分中，我们只讨论2MiB的大页，因为 1GiB 的页相当少见，对于我们的任务来说纯属浪费。

架构	最小页大小	大页大小
x86	4KiB	2MiB, 4MiB
x86-64	4KiB	2MiB, 1GiB[20]
ARMv7	4KiB	64KiB, 1MiB, 16MiB
ARMv8	4KiB	16KiB, 64KiB
RISCV32	4KiB	4MiB
RISCV64	4KiB	2MiB, 1GiB, 512GiB, 256TiB
Power ISA	8KiB	64KiB, 16MiB, 16GiB

当今常用架构中的页大小，来自维基百科

大页的主要优势在于管理成本更低，因为覆盖相同内存量所需的页更少。此外其他操作的成本也更低，例如将虚拟地址解析为物理地址，因为所需要的页表少了一级：以一个 21 位的偏移量代替页中原来的12位偏移量，从而少一级页表。

这减轻了处理此转换的 CPU 部分的压力，因而在许多情况下提高了性能。但是在我们的例子中，压力不在于遍历页表的硬件，而在内核中运行的软件。

在 Linux 上，我们可以通过多种方式分配 2MiB 大页，例如分配与 2MiB 对齐的内存，然后使用 madvise 告诉内核为提供的缓冲区使用大页：

```
void* buf = aligned_alloc(1 << 21, size);
madvise(buf, size, MADV_HUGEPAGE)
```

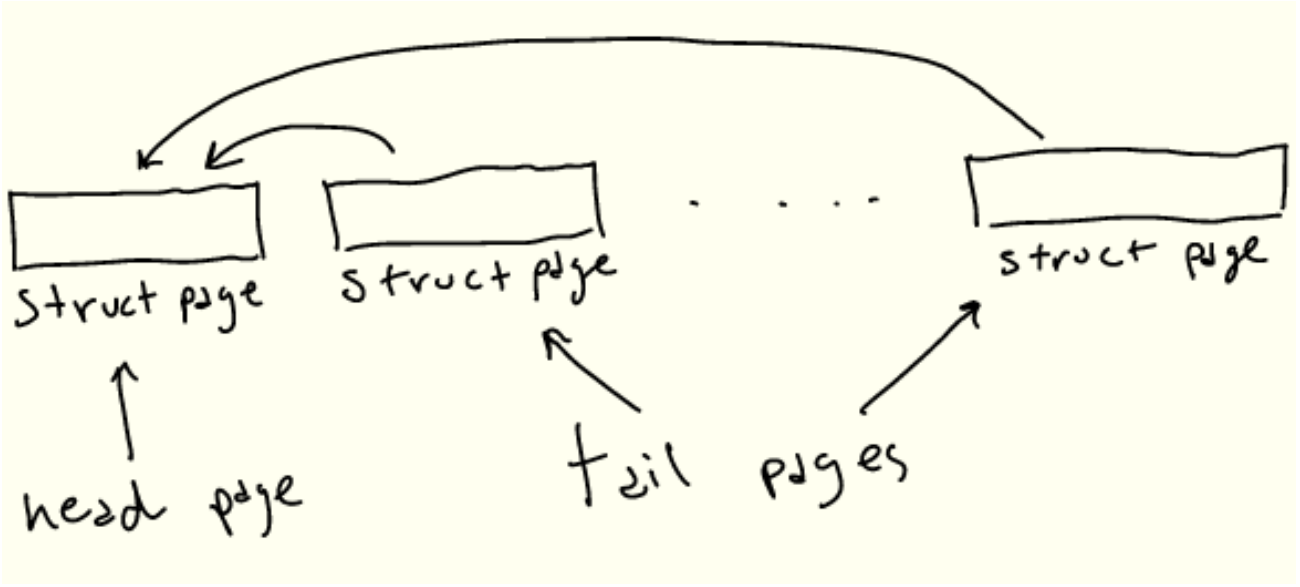
切换到大页又给我们的程序带来了约 50% 的性能提升：

```
% ./write --write_with_vmsplice --huge_page | ./read --reac
51.0GiB/s, 256KiB buffer, 40960 iterations (10GiB piped)
```

然而，提升的原因并不完全显而易见。我们可能会天真地认为，通过使用大页， struct page 将只引用 2MiB 页，而不是 4KiB 页面。

遗憾的是，情况并非如此：内核代码假定 struct page 引用当前架构的“标准”大小的页。这种实现作用于大页（通常Linux称之为“复合页面”）的方式是，“头” struct page 包含关于背后物理页的实际信息，而连续的“尾”页仅包含指向头页的指针。

因此为了表示 2MiB 的大页，将有1个“头”struct page，最多 511 个“尾”struct pages。或者对于我们的 128KiB 缓冲区来说，有 31个尾 struct pages：



Busy looping

我们很快就要完成了，我保证！再看一下 perf 的输出：

```
-   46.91%     0.38%  write    libc-2.33.so      [.] vmsplice
  - 46.84% vmsplice
     - 43.15% entry_SYSCALL_64_after_hwframe
        - do_syscall_64
           - 41.80% __do_sys_vmsplice
              + 14.90% wait_for_space
              + 8.27% __wake_up_common_lock
                4.40% add_to_pipe
              + 4.24% iov_iter_get_pages
              + 3.92% __mutex_lock.constprop.0
                1.81% iov_iter_advance
              + 0.55% import_iovec
           + 0.76% syscall_exit_to_user_mode
        1.54% syscall_return_via_sysret
        1.49% __entry_text_start
```

现在大量时间花费在等待管道可写（wait_for_space），以及唤醒等待管道填充内容的读程序（__wake_up_common_lock）。

为了避免这些同步成本，如果管道无法写入，我们可以让 vmsplice 返回，并执行忙循环直到写入为止——在用 splice 读取时做同样的处理：

```
...
// SPLICE_F_NONBLOCK will cause `vmsplice` to return ir
// if we can't write to the pipe, returning EAGAIN
ssize_t ret = vmsplice(STDOUT_FILENO, &bufvec, 1, SPLIC
if (ret < 0 && errno == EAGAIN) {
  continue; // busy loop if not ready to write
}
...
```

通过忙循环，我们的性能又提高了25%：

```
% ./write --write_with_vmsplice --huge_page --busy_loop |
62.5GiB/s, 256KiB buffer, 40960 iterations (10GiB piped)
```

总结

通过查看 perf 输出和 Linux 源码，我们系统地提高了程序性能。在高性能编程方面，管道和拼接并不是真正的热门话题，而我们这里所涉及的主题是：零拷贝操作、环形缓冲区、分页与虚拟内存、同步开销。

尽管我省略了一些细节和有趣的话题，但这篇博文还是已经失控而变得太长了：

- 在实际代码中，缓冲区是分开分配的，通过将它们放置在不同的页表条目中来减少页表争用（FizzBuzz程序也是这样做的）。

- 记住，当使用 get_user_pages 获取页表条目时，其 refcount 增加，而 put_page 减少。如果我们为两个缓冲区使用两个页表条目，而不是为两个缓冲器共用一个页表条目的话，那么在修改 refcount 时争用更少。
- 通过用taskset将./write和./read进程绑定到两个核来运行测试。
- 资料库中的代码包含了我试过的许多其他选项，但由于这些选项无关紧要或不够有趣，所以最终没有讨论。
- 资料库中还包含get_user_pages_fast 的一个综合基准测试，可用来精确测量在用或不用大页的情况下运行的速度慢多少。
- 一般来说，拼接是一个有点可疑/危险的概念，它继续困扰着内核开发人员。

请让我知道本文是否有用、有趣或不一定！

致谢

非常感谢 Alexandru Scvorţov、Max Staudt、Alex Appetiti、Alex Sayers、Stephen Lavelle、Peter Cawley和Niklas Hambüchen审阅了本文的草稿。Max Staudt 还帮助我理解了 get_user_page 的一些微妙之处。

1. 这将在风格上类似于我的atan2f性能调研（https://mazzo.li/posts/vectorized-atan2.html），尽管所讨论的程序仅用于学习。此外，我们将在不同级别上优化代码。调优 atan2f 是在汇编语言输出指导下进行的微优化，调优管道程序则涉及查看 perf 事件，并减少各种内核开销。

2. 本测试在英特尔 Skylake i7-8550U CPU 和 Linux 5.17 上运行。你的环境可能会有所不同，因为在过去几年中，影响本文所述程序的 Linux 内部结构一直在不断变化，并且在未来版本中可能还会调整。

3. “FizzBuzz”据称是一个常见的编码面试问题，虽然我个人从来没有被问到过该问题，但我有确实发生过的可靠证据。

4. 尽管我们固定了缓冲区大小，但即便我们使用不同的缓冲区大小，考虑到其他瓶颈的影响，（吞吐量）数字实际也并不会有很大差异。

5. 关于有趣的细节，可随时参考资料库。一般来说，我不会在这里逐字复制代码，因为细节并不重要。相反，我将贴出更新的代码片段。

6. 注意，这里我们分析了一个包括管道读取和写入的shell调用——默认情况下，perf record跟踪所有子进程。

7. 分析该程序时，我注意到 perf 的输出被来自“Pressure Stall Information”基础架构（PSI）的信息所污染。因此这些数字取自一个禁用PSI后编译的内核。这可以通过在内核构建配置中设置 CONFIG_PSI=n 来实现。在NixOS 中：

```
boot.kernelPatches = [{
  name = "disable-psi";
  patch = null;
  extraConfig = "
    PSI n
  ";
}];
```

此外，为了让 perf 能正确显示在系统调用中花费的时间，必须有内核调试符号。如何安装符号因发行版而异。在最新的 NixOS 版本中，默认情况下会安装它们。

8. 假如你运行了 perf record -g，可以在 perf report 中用 + 展开调用图。

9. 被称为 tmp_page 的单一“备用页”实际上由 pipe_read 保留，并由pipe_write 重用。然而由于这里始终只是一个页面，我无法利用它来实现更高的性能，因为在调用 pipe_write 和 pipe_ read 时，页面重用会被固定开销所抵消。

10. 从技术上讲，vmsplice 还支持在另一个方向上传输数据，尽管用处不大。如手册页所述：

> vmsplice实际上只支持从用户内存到管道的真正拼接。反方向上，它实际上只是将数据复制到用户空间。

11. Travis Downs 指出，该方案可能仍然不安全，因为页面可能会被进一步拼接，从而延长其生命期。这个问题也出现在最初的 FizzBuzz 帖子中。事实上，我并不完全清楚不带 SPLICE_F_GIFT 的 vmsplice 是否真的不安全——vmsplic 的手册页说明它是安全的。然而，在这种情况下，绝对需要特别小心，以实现零拷贝管道，同时保持安全。在测试程序中，读取端将管道拼接到/dev/null 中，因此可能内核知道可以在不复制的情况下拼接页面，但我尚未验证这是否是实际发生的情况。

12. 这里我们呈现了一个简化模型，其中物理内存是一个简单的平面线性序列。现实情况复杂一些，但简单模型已能说明问题。

13. 可以通过读取 /proc/self/pagemap 来检查分配给当前进程的虚拟页面所对应的物理地址，并将“页面帧号”乘以页面大小。

14. 从 Ice Lake 开始，英特尔将页表扩展为5级，从而将最大可寻址内存从256TiB 增加到 128PiB。但此功能必须显式开启，因为有些程序依赖于指针的高 16 位不被使用。

15. 页表中的地址必须是物理地址，否则我们会陷入死循环。

16. 注意，高 16 位未使用：这意味着我们每个进程最多可以处理 248 – 1 字节，或 256TiB 的物理内存。

17. struct page 可能指向尚未分配的物理页，这些物理页还没有物理地址和其他与页相关的抽象。它们被视为对物理页面的完全抽象的引用，但不一定是对已分配的物理页面的引用。这一微妙观点将在后面的旁注中予以说明。

18. 实际上，管道代码总是在 nr_pages = 16 的情况下调用 get_user_pages_fast，必要时进行循环，这可能是为了使用一个小的静态缓冲区。但这是一个实现细节，拼接页面的总数仍将是32。

19. 以下部分是本文不需要理解的微妙之处！

如果页表不包含我们要查找的条目，get_user_pages_fast 仍然需要返回一个 struct page。最明显的方法是创建正确的页表条目，然后返回相应的 struct page。

然而，get_user_pages_fast 仅在被要求获取 struct page 以写入其中时才会这样做。否则它不会更新页表，而是返回一个 struct page，给我们一个尚未分配的物理页的引用。这正是 vmsplice 的情况，因为我们只需要生成一个 struct page 来填充管道，而不需要实际写入任何内存。

换句话说，页面分配会被延迟，直到我们实际需要时。这节省了分配物理页的时间，但如果页面从未通过其他方式填充错误，则会导致重复调用 get_user_pages_fast 的慢路径。

因此，如果我们之前不进行 memset，就不会“手动”将错误页放入页表中，结果是不仅在第一次调用 get_user_pages_fast 时会陷入慢路径，而且所有后续调用都会出现慢路径，从而导致明显地减速（25GiB/s而不是30GiB/s）：

```
% ./write --write_with_vmsplice --dont_touch_pages | ./rea
25.0GiB/s, 256KiB buffer, 40960 iterations (10GiB piped)
```

此外，在使用大页时，这种行为不会表现出来：在这种情况下，get_user_pages_fast 在传入一系列虚拟内存时，大页支持将正确地填充错误页。

如果这一切都很混乱，不要担心，get_user_page 和 friends 似乎是内核中非常棘手的一角，即使对于内核开发人员也是如此。

20. 仅当 CPU 具有 PDPE1GB 标志时。

21. 例如，CPU包含专用硬件来缓存部分页表，即“转换后备缓冲区”（translation lookaside buffer，TLB）。TLB 在每次上下文切换（每次更改 CR3 的内容）时被刷新。大页可以显著减少 TLB 未命中，因为 2MiB 页的单个条目覆盖的内存是 4KiB 页面的 512 倍。

22. 如果你在想“太烂了！”正在进行各种努力来简化和/或优化这种情况。最近的内核（从5.17开始）包含了一种新的类型，struct folio，用于显式标识头页。这减少了运行时检查 struct page 是头页还是尾页的需要，从而提高了性能。其他努力的目标是彻底移除额外的 struct pages，尽管我不知道怎么做的。

本文文字及图片出自 CSDN

分享这篇文章：

相关文章：

1. 系统管理员喜欢 systemd 的 5 个理由
2. 小米工程师提交优化补丁被批，Linux内核维护者：太疯狂！
3. 在 Linux 中使用组合键输出特殊字符
4. 10 款更先进的开源命令行工具
5. Debian 侵犯了 Rust 的商标权，这将导致一个极其严重的后果
6. 时隔半年，Linux性能重新超越Windows 11
7. 12 岁印度少年出手，Ubuntu Unity 桌面 6 年后“起死回生”！
8. Linux内核将引入Rust，Linus：以防此事搞砸了我又发脾气，先给大家道个歉
9. 在 Linux 上使用 Bash 创建一个临时文件 | Linux 中国

10. Linux之父发话：Rust即将出现在Linux内核中

你的反应是：

请关注我们：