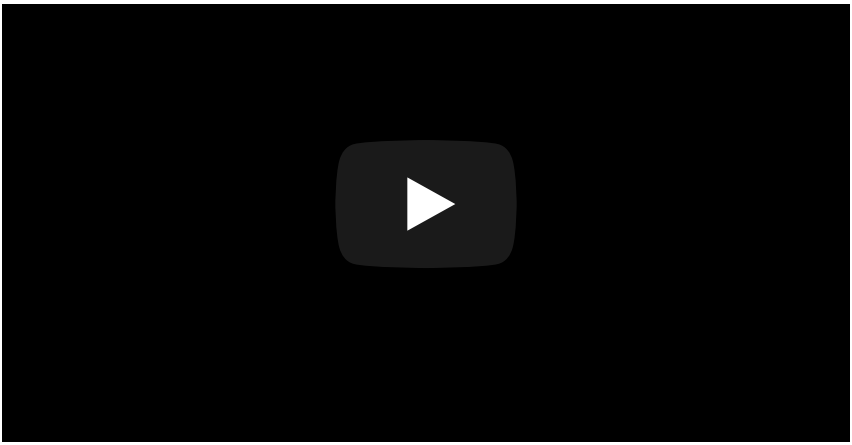
关于 swap 的一些补充

上周翻译完 【译】替 swap 辩护：常见的误解 之后很多朋友们似乎还有些疑问和误解，于是写篇后续澄清一下。事先声明我不是内核开发者， 这里说的只是我的理解， 基于内核文档中关于物理内存的描述 ，新的内核代码的具体行为可能和我的理解有所出入，欢迎踊跃讨论。

Introduction to Memory Management in Linux



误解1: swap 是虚拟内存，虚拟内存肯定比物理内存慢嘛

这种误解进一步的结论通常是：「使用虚拟内存肯定会减慢系统运行时性能，如果物理内存足够为什么还要用虚拟的？」 这种误解是把虚拟内存和交换区的实现方式类比于「虚拟磁盘」或者「虚拟机」等同的方式，也隐含「先用物理内存，用完了之后用虚拟内存」也即下面的「误解3」的理解。

首先，交换区（swap） **不是** 虚拟内存。操作系统中说「物理内存」还是「虚拟内存」的时候在指程序代码 寻址时使用的内存地址方式，使用物理地址空间时是

在访问物理内存，使用虚拟地址空间时是在访问虚拟内存。 现代操作系统在大部分情况下都在使用虚拟地址空间寻址， **包括** 在执行内核代码的时候。

并且，交换区 **不是** 实现虚拟内存的方式。操作系统使用内存管理单元（MMU，Memory Management Unit）做虚拟内存地址到物理内存地址的地址翻译，现代架构下 MMU 通常是 CPU 的一部分，配有它专用的一小块存储区叫做地址转换旁路缓存（TLB，Translation Lookaside Buffer）， 只有在 TLB 中没有相关地址翻译信息的时候 MMU 才会以缺页中断的形式调用操作系统内核帮忙。 除了 TLB 信息不足的时候，大部分情况下使用虚拟内存都是硬件直接实现的地址翻译，没有软件模拟开销。 实现虚拟内存不需要用到交换区，交换区只是操作系统实现虚拟内存后能提供的一个附加功能， 即便没有交换区，操作系统大部分时候也在用虚拟内存，包括在大部分内核代码中。

误解2: 但是没有交换区的话，虚拟内存地址都有物理内存对应嘛

很多朋友也理解上述操作系统实现虚拟内存的方式，但是仍然会有疑问：「我知道虚拟内存和交换区的区别， 但是没有交换区的话，虚拟内存地址都有物理内存对应，不用交换区的话就不会遇到读虚拟内存需要读写磁盘 导致的卡顿了嘛」。

这种理解也是错的，禁用交换区的时候，也会有一部分分配给程序的虚拟内存不对应物理内存， 比如使用 `mmap` 调用实现内存映射文件的时候。实际上即便是使用 `read/write` 读写文件， Linux 内核中（可能现代操作系统内核都）在底下是用和 `mmap` 相同的机制建立文件 到虚拟地址空间的地址映射，然后实际读写到虚拟地址时靠缺页中断把文件内容载入页面缓存（page cache）。内核加载可执行程序和动态链接库的方式也是通过内存映射文件。甚至可以进一步说， 用户空间的虚拟内存地址范围内，除了匿名页之外，其它虚拟地址都是文件后备（backed by file），而匿名页通过交换区作为文件后备。上篇文章中提到的别的类型的内存，比如共享内存页面（shm）是被一个内存中的虚拟文件系统后备的，这一点有些套娃先暂且不提。于是事实是无论有没有交换区， 缺页的时候总会有磁盘读写从慢速存储加载到物理内存，这进一步引出上篇文章中对于交换区和页面缓存这两者的讨论。

误解3： 不是内存快用完的时候才会交换的么？

简短的答案可以说「是」，但是内核理解的「内存快用完」和你理解的很可能不同。 也可以说「不是」，就算按照内核理解的「内存快用完」的定义，内存快用完的时候内核的行为是去回收内存， 至于回收内存的时候内核会做什么有个复杂的启发式经验算法，实际上真的内存快满的时候根本来不及做 swap ，内核可能会尝试丢弃 page cache 甚至丢弃 vfs cache (dentry cache / inode cache) 这些不需要磁盘I/O就能更快获取可用内存的动作。

深究这些内核机制之前，我在思考为什么很多朋友会问出这样的问题。可能大部分这么问的人，学过编程， 稍微学过基本的操作系统原理，在脑海里对内核分配页面留着这样一种印象（C伪代码）：

```
1  ////////////////////  userspace space  ////////////////
2  void* malloc(int size){
3      void* pages = mmap(...);                    // 从内核分配内存页
4      return alloc_from_page(pages, size);        // 从拿到的内存页细分
5  }
```

```
 6
 7  ////////////////////  kernel space
//////////////////
 8  void * SYSCALL do_mmap(...){
 9     //...
10     return kmalloc_pages(nr_page);
11  }
12
13  void* kmalloc_pages(int size){
14    while ( available_mem < size ) {
15      // 可用内存不够了！尝试搞点内存
16      page_frame_info* least_accessed =
 lru_pop_page_frame();     // 找出最少访
问的页面
17      switch ( least_accessed -> pf_ty
pe ){
18        case PAGE_CACHE: drop_page_cac
he(least_accessed); break;  // 丢弃文件缓存

19        case SWAP:       swap_out(leas
t_accessed);        break;  // <- 写磁盘
，所以系统卡了！
20        // ... 别的方式回收 least_access
ed
21      }
22      append_free_page(free_page_list,
least_accessed);           // 回收到的页
面加入可用列表
23      available_mem += least_accessed
-> size;
24    }
25    // 搞到内存了！返回给程序
```

```
26    available_mem -= size;
27    void * phy_addr = take_from_free_l
ist(free_page_list, size);
28    return assign_virtual_addr(phy_addr
);
29  }
```

这种逻辑隐含三层 **错误的** 假设：

1. 分配物理内存是发生在从内核分配内存的时候的，比如 `malloc/mmap` 的时候。
2. 内存回收是发生在进程请求内存分配的上下文里的，换句话说进程在等内核的内存回收返回内存，不回收到内存，进程就得不到内存。
3. 交换出内存到 swap 是发生在内存回收的时候的，会阻塞内核的内存回收，进而阻塞程序的内存分配。

这种把内核代码当作「具有特权的库函数调用」的看法，可能很易于理解， 甚至早期可能的确有操作系统的内核是这么实现的，但是很可惜现代操作系统都不是这么做的。 上面三层假设的错误之处在于：

1. 在程序请求内存的时候，比如 `malloc/mmap` 的时候，内核只做虚拟地址分配， 记录下某段虚拟地址空间对这个程序是可以合法访问的，但是不实际分配物理内存给程序。 在程序第一次访问到虚拟地址的时候，才会实际分配物理内存。这种叫 **惰性分配（lazy allocation）** 。

2. 在内核感受到内存分配压力之后，早在内核内存用尽之前，内核就会在后台慢慢扫描并回收内存页。内存回收通常不发生在内存分配的时候，除非在内存非常短缺的情况下，后台内存回收来不及满足当前分配请求， 才会发生 **直接回收(direct reclamation)** 。
3. 同样除了直接回收的情况，大部分正常情况下换出页面是内存管理子系统调用 DMA 在后台慢慢做的， 交换页面出去不会阻塞内核的内存回收，更不会阻塞程序做内存分配（malloc ）和使用内存(实际访问惰性分配的内存页)。

也就是说，现代操作系统内核是高度并行化的设计，内存分配方方面面需要消耗计算资源或者 I/O 带宽的场景，都会尽量并行化，最大程度利用好计算机所有组件（CPU/MMU/DMA/IO）的吞吐率， 不到紧要关头需要直接回收的场合，就不会阻塞程序的正常执行流程。

惰性分配有什么好处？

或许会有人问：「我让你分配内存，你给我分配了个虚拟的，到用的时候还要做很多事情才能给我，这不是骗人嘛」， 或者会有人担心惰性分配会对性能造成负面影响。

这里实际情况是程序从分配虚拟内存的时候，「到用的时候」，这之间有段时间间隔，可以留给内核做准备 。程序可能一下子分配一大片内存地址，然后再在执行过程中解析数据慢慢往地址范围内写东西。 程序分配虚拟内存的速率可以是「突发」的，比如一个系统调用中分配 1GiB 大小，而实际写入数据的速率会被 CPU 执行速度等因素限制，不会短期内突然写入很多页面。 这个分配速率导致的时间差内内核可以完成很多后台工作，比如回收内存， 比如把回收到的别的进程用过的内存页面初始化为全0，这部分后台工作可以和程序的执行过程并行， 从而当程序实际用到内存的时候，需要的准备工作已经做完了，大部分场景下可以直接分配物理内存出来。

如果程序要做实时响应，想避免因为惰性分配造成的性能不稳定，可以使用 `mlock/mlockall` 将得到的虚拟内存锁定在物理内存中，锁的过程中内核会做物理内存分配。不过要区分「性能不稳定」和「低性能」， 预先分配内存可以避免实际使用内存时分配物理页面的额外开销，但是会拖慢整体吞吐率，所以要谨慎使用。

很多程序分配了很大一片地址空间，但是实际并不会用完这些地址，直到程序执行结束这些虚拟地址也一直 处于没有对应物理地址的情况。惰性分配可以避免为这些情况浪费物理内存页面，使得很多程序可以无忧无虑地 随意分配内存地址而不用担心性能损失。这种分配方式也叫「超额分配（overcommit）」。飞机票有超

售， VPS 提供商划分虚拟机有超售，操作系统管理内存也同样有这种现象，合理使用超额分配能改善整体系统效率。

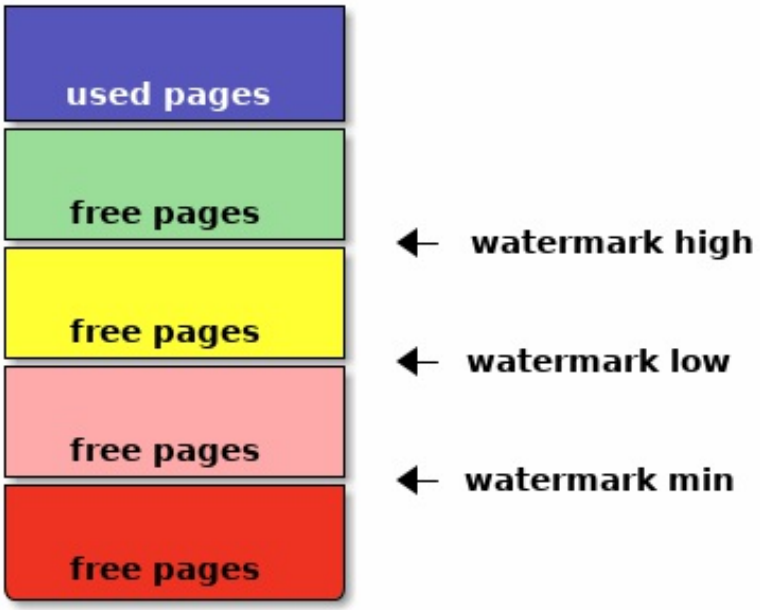
内核要高效地做到惰性分配而不影响程序执行效率的前提之一，在于程序真的用到内存的时候， 内核能不做太多操作就立刻分配出来，也就是说内核需要时时刻刻在手上留有一部分空页， 满足程序执行时内存分配的需要。换句话说，内核需要早在物理内存用尽之前，就开始回收内存。

那么内核什么时候会开始回收内存？

首先一些背景知识：物理内存地址空间并不是都平等，因为一些地址范围可以做 DMA 而另一些不能，以及 NUMA 等硬件环境倾向于让 CPU 访问其所在 NUMA 节点内存范围。在 32bit 系统上内核的虚拟地址空间还有低端内存和高端内存的区分，他们会倾向于使用不同属性的物理内存，到 64bit 系统上已经没有了这种限制。

硬件限制了内存分配的自由度，于是内核把物理内存空间分成多个 Zone ，每个 Zone 内各自管理可用内存， Zone 内的内存页之间是相互平等的。

zone 内水位线



一个 Zone 内的页面分配情况可以右图描绘。 除了已用内存页，剩下的就是空闲页（free pages），空闲页范围中有三个水位线（watermark）评估当前内存压力情况，分别是高位（high）、低位（low）、最小位（min）。

当内存分配使得空闲页水位低于低位线，内核会唤醒 kswapd 后台线程， kswapd 负责扫描物理页面的使用情况并挑选一部分页面做回收，直到可用页面数量恢复到水位线高位（high）以上。 如果 kswapd 回收内存

的速度慢于程序执行实际分配内存的速度， 可用空闲页数量可能进一步下降，降至低于最小水位（min）之后，内核会让内存分配进入 **直接回收(direct reclamation)** 模式，在直接回收模式下，程序分配某个物理页的请求（ 第一次访问某个已分配虚拟页面的时候）会导致在进程上下文中阻塞式地调用内存回收代码。

除了内核在后台回收内存，进程也可以主动释放内存，比如有程序退出的时候就会释放一大片内存页， 所以可用页面数量可能会升至水位线高位以上。有太多可用页面浪费资源对整体系统运行效率也不是好事， 所以系统会积极缓存文件读写，所有 page cache 都留在内存中，直到可用页面降至低水位以下触发 `kswapd` 开始工作。

设置最小水位线（min）的原因在于，内核中有些硬件也会突然请求大量内存，比如来自网卡接收到的数据包， 预留出最小水位线以下的内存给内核内部和硬件使用。

设置高低两个控制 `kswapd` 开关的水位线是基于控制理论。唤醒 `kswapd` 扫描内存页面本身有一定计算开销，于是每次唤醒它干活的话就让它多做一些活（ high - low ），避免频繁多次唤醒。

因为有这些水位线，系统中根据程序请求内存的「速率」，整个系统的内存分配在宏观的一段时间内可能处于以下几种状态：

1. **不回收：** 系统中的程序申请内存速度很慢，或者程序主动释放内存的速度很快， （比如程序执行

时间很短，不怎么进行文件读写就马上退出，）此时可用页面数量可能一直处于低水位线以上， 内核不会主动回收内存，所有文件读写都会以页面缓存的形式留在物理内存中。

2. **后台回收：** 系统中的程序在缓慢申请内存，比如做文件读写， 比如分配并使用匿名页面。系统会时不时地唤醒 `kswapd` 在后台做内存回收， 不会干扰到程序的执行效率。
3. **直接回收：** 如果程序申请内存的速度快于 `kswapd` 后台回收内存的速度， 空闲内存最终会跌破最小水位线，随后的内存申请会进入直接回收的代码路径，从而极大限制内存分配速度。 在直接分配和后台回收的同时作用下，空闲内存可能会时不时回到最小水位线以上， 但是如果程序继续申请内存，空闲内存量就会在最小水位线附近上下徘徊。
4. **杀进程回收：** 甚至直接分配和后台回收的同时作用也不足以拖慢程序分配内存的速度的时候， 最终空闲内存会完全用完，此时触发 OOM 杀手干活杀进程。

系统状态处于 **1. 不回收** 的时候表明分配给系统的内存量过多，比如系统刚刚启动之类的时候。 理想上应该让系统长期处于 **2. 后台回收** 的状态，此时最大化利用缓存的效率而又不会因为内存回收 减缓程序执行速度。如果系统引导后长期处于 **1. 不回收** 的状态下，那么说明没有充分利用空闲内存做 文件缓存，有些 unix 服务比如 preload 可用来提前填充文件缓存。

如果系统频繁进入 **3. 直接回收** 的状态，表明在这种工作负载下系统需要减慢一些内存分配速度， 让 kswapd 有足够时间回收内存。就如前一篇翻译中 Chris 所述，频繁进入这种状态也不一定代表「内存不足」，可能表示内存分配处于非常高效的利用状态下， 系统充分利用慢速的磁盘带宽，为快速的内存缓存提供足够的可用空间。 **直接回收** 是否对进程负载有负面影响要看具体负载的特性。 此时选择禁用 swap 并不能降低磁盘 I/O，反而可能缩短 **2. 后台回收** 状态能持续的时间， 导致更快进入 **4. 杀进程回收** 的极端状态。

当然如果系统长期处于 **直接回收** 的状态的话，则说明内存总量不足，需要考虑增加物理内存， 或者减少系统负载了。如果系统进入 **4. 杀进程回收** 的状态，不光用空间的进程会受影响， 并且还可能导致内核态的内存分配受影响，产生网络丢包之类的结果。

微调内存管理水位线

可以看一下运行中的系统中每个 Zone 的水位线在哪儿。比如我手上这个 16GiB 的系统中：

```
$ cat /proc/zoneinfo
Node 0, zone      DMA
   pages free     3459
         min      16
         low      20
         high     24
         spanned  4095
         present  3997
         managed  3975
Node 0, zone    DMA32
   pages free     225265
         min      3140
         low      3925
         high     4710
         spanned  1044480
         present  780044
         managed  763629
Node 0, zone   Normal
   pages free     300413
         min      13739
         low      17173
         high     20607
         spanned  3407872
         present  3407872
         managed  3328410
```

因为不是 NUMA 系统，所以只有一个 NUMA node，其中根据 DMA 类型共有 3 个 Zone 分别叫 DMA, DMA32, Normal 。三个 Zone 的物理地址范围（spanned）加起来大概有 \(4095+1044480+3407872\)

大约 17GiB 的地址空间，而实际可访问的地址范围（present）加起来有 \(3997+780044+3407872\) 大约 16GiB 的可访问物理内存。

其中空闲页面有 \(3459+762569+1460218\) 大约 8.5GiB，三条水位线分别在：\(\texttt{high} = 24+4710+20607 = 98\texttt{MiB}\)，\(\texttt{low} = 20+3925+17173 = 82\texttt{MiB}\)，\(\texttt{min} = 16+3140+13739 = 65\texttt{MiB}\) 的位置。

具体这些水位线的确定方式基于几个 sysctl。首先 min 基于 `vm.min_free_kbytes` 默认是基于内核低端内存量的平方根算的值，并限制到最大 64MiB 再加点余量，比如我这台机器上 `vm.min_free_kbytes = 67584`，于是 min 水位线在这个位置。其它两个水位线基于这个计算，在 min 基础上增加总内存量的 `vm.watermark_scale_factor / 10000` 比例（在小内存的系统上还有额外考虑），默认 `vm.watermark_scale_factor = 10` 在大内存系统上意味着 low 比 min 高 0.1%，high 比 low 高 0.1%。

可以手动设置这些值，以更早触发内存回收，比如将 `vm.watermark_scale_factor` 设为 100:

```
 1  $ echo 100 | sudo tee /proc/sys/vm/w
atermark_scale_factor
 2  $ cat /proc/zoneinfo
 3  Node 0, zone      DMA
 4      pages free     3459
 5            min      16
 6            low      55
 7            high     94
 8            spanned  4095
 9            present  3997
10            managed  3975
11      Node 0, zone    DMA32
12      pages free     101987
13            min      3149
14            low      10785
15            high     18421
16            spanned  1044480
17            present  780044
18            managed  763629
19      Node 0, zone   Normal
20      pages free     61987
21            min      13729
22            low      47013
23            high     80297
24            spanned  3407872
25            present  3407872
26            managed  3328410
```

得到的三条水位线分别在 $\texttt{min} = 16+3149+13729 = 66\texttt{MiB}$ ， $\texttt{low} = 55+10785+47013 = 226\texttt{MiB}$ ， $\texttt{high} = 94+18421+80297 = 386\texttt{MiB}$ ， 从而 low 和 high 分别比 min 提高 160MiB 也就是内存总量的 1% 左右。

在 swap 放在 HDD 的系统中，因为换页出去的速度较慢，除了上篇文章说的降低 `vm.swappiness` 之外，还可以适当提高 `vm.watermark_scale_factor` 让内核更早开始回收内存，这虽然会稍微降低缓存命中率，但是另一方面可以在进入直接回收模式之前 有更多时间做后台换页，也将有助于改善系统整体流畅度。

只有 0.1%，这不就是说内存快用完的时候么？

所以之前的「误解3」我说答案可以说「是」或者「不是」，但是无论回答是或不是，都代表了认为「swap 就是额外的慢速内存」的错误看法。当有人在强调「swap 是内存快用完的时候才交换」的时候， 隐含地，是在把系统总体的内存分配看作是一个静态的划分过程：打个比方这就像在说，我的系统里存储空间有快速 128GiB SSD 和慢速 HDD 的 1TiB ，同样内存有快速的 16GiB RAM 和慢速 16GiB 的 swap 。 这种静态划分的

类比是错误的看待方式，因为系统回收内存进而做页面交换的方式是动态平衡的过程， 需要考虑到「时间」和「速率」而非单纯看「容量」。

假设 swap 所在的存储设备可以支持 5MiB/s 的吞吐率（ HDD 上可能更慢， SSD 上可能更快，这里需要关注数量级），相比之下 DDR3 大概有 10GiB/s 的吞吐率，DDR4 大概有 20GiB/s ，无论多快的 SSD 也远达不到这样的吞吐（可能 Intel Optane 这样的 DAX 设备会改变这里的状况）。从而把 swap 当作慢速内存的视角来看的话，加权平均的速率是非常悲观的，「 16G 的 DDR3 + 16G 的 swap 会有 $\frac{16 \times 10 \times 1024 + 16 \times 5}{16+16} = 5 \texttt{GiB/s}$ 的吞吐？所以开 swap 导致系统速度降了一半？」显然不能这样看待。

动态的看待方式是， swap 设备能提供 5MiB/s 的吞吐，这意味着：如果我们能把未来 10 分钟内不会访问到的页面换出到 swap ，那么就相当于有 $10 \times 60 \texttt{s} \times 5 \texttt{MiB/s} = 3000 \texttt{MiB}$ 的额外内存，用来放那 10 分钟内可能会访问到的页面缓存。 10 分钟只是随口说的一段时间，可以换成 10 秒或者 10 小时，重要的是只要页面交换发生在后台， 不阻塞前台程序的执行，那么 swap 设备提供的额外吞吐率相当于一段时间内提供了更大的物理内存， 总是能提升页面缓存的命中，从而改善系统性能。

当然系统内核不能预知「未来 10 分钟内需要的页面」，只能根据历史上访问内存的情况预估之后可能会访问的情况， 估算不准的情况下，比如最近10分钟内用过的页面缓存在之后10分钟内不再被使用的时候， 为了

把最近这10分钟内访问过的页面留在物理内存中，可能会把之后10分钟内要用到的匿名页面换出到了交换设备上。 于是会有下面的情况：

但是我开了 swap 之后，一旦复制大文件，系统就变卡，不开 swap 不会这样的

大概电脑用户都经历过这种现象，不限于 Linux 用户，包括 macOS 和 Windows 上也是。 在文件管理器中复制了几个大文件之后，切换到别的程序系统就极其卡顿，复制已经结束之后的一段时间也会如此。 复制的过程中系统交换区的使用率在上涨，复制结束后下降，显然 swap 在其中有重要因素，并且禁用 swap 或者调低 swappiness 之后就不会这样了。于是网上大量流传着解释这一现象，并进一步建议禁用 swap 或者调低 swappiness 的文章。我相信不少关心系统性能调优的人看过这篇「Tales from responsivenessland: why Linux feels slow, and how to fix that」或是它的转载、翻译，用中文搜索的话还能找到更多错误解释 swappiness 目的的文章，比如 这篇将 swappiness 解释成是控制内存和交换区比例的参数 。

除去那些有技术上谬误的文章，这些网文中描述的现象是有道理的，不单纯是以讹传讹。 桌面环境中内存分配策略的不确定性和服务器环境中很不一样，复制、下载、解压大文件等导致一段时间内 大量占用页面缓存，以至于把操作结束后需要的页面撵出物理内存，无论是交换出去的方式还是以丢弃页面缓存的方式， 都会导致桌面响应性降低。

不过就像前文 Chris 所述，这种现象其实并不能通过禁止 swap 的方式缓解：禁止 swap 或者调整 swappiness 让系统尽量避免 swap 只影响回收匿名页面的策略，不影响系统回收页面的时机， 也不能避免系统丢弃将要使用的页面缓存而导致的卡顿。

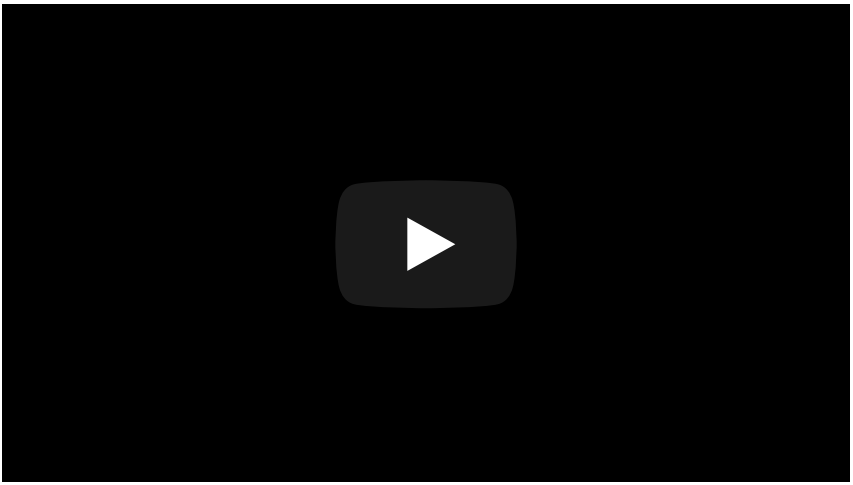
以前在 Linux 上也没有什么好方法能避免这种现象。 macOS 转用 APFS 作为默认文件系统之后， 从文件管理器（Finder）复制文件默认启用 file clone 快速完成，这操作不实际复制文件数据， 一个隐含优势在不需要读入文件内容，从而不会导致大量页面缓存失效。 Linux 上同样可以用支持 reflink 的文件系统比如 btrfs 或者开了 reflink=1 的 xfs 达到类似的效果。 不过 reflink 也只能拯救复制文件的情况，不能改善解压文件、下载文件、计算文件校验等情况下， 一次性处理大文件对内存产生的压力。

好在最近几年 Linux 有了 cgroup ，允许更细粒度地调整系统资源分配。进一步现在我们有了 cgroup v2 ，前面 Chris 的文章也有提到 cgroup v2 的 `memory.low` 可以某种程度上建议内存子系统 尽量避免回收某些 cgroup 进程的内存。

于是有了 cgroup 之后，另一种思路是把复制文件等大量使用内存而之后又不需要保留页面缓存的程序单独放入 cgroup 内限制它的内存用量，用一点点复制文件时的性能损失换来整体系统的响应流畅度。

关于 cgroup v1 和 v2

稍微跑题说一下 cgroup v2 相对于 v1 带来的优势。这方面优势在 Chris Down 另一个关于 cgroup v2 演讲 中有提到。老 cgroup v1 按控制器区分 cgroup 层级，从而内存控制器所限制的东西和 IO 控制器所限制的东西是独立的。在内核角度来看，页面写回（page writeback）和交换（swap）正是夹在内存控制器和IO控制器管理的边界上，从而用 v1 的 cgroup 难以同时管理。 v2 通过统一控制器层级解决了这方面限制。具体见下面 Chris Down 的演讲。



用 cgroup v2 限制进程的内存分配

实际上有了 cgroup v2 之后，还有更多控制内存分配的方案。 cgroup v2 的内存控制器 可以对某个 cgroup 设置这些阈值：

- **memory.min** : 最小内存限制。内存用量低于此值后系统不会回收内存。
- **memory.low** : 低内存水位。内存用量低于此值后系统会尽量避免回收内存。
- **memory.high** : 高内存水位。内存用量高于此值后系统会积极回收内存，并且会对内存分配节流（throttle）。
- **memory.max** : 最大内存限制。内存用量高于此值后系统会对内存分配请求返回 ENOMEM，或者在 cgroup 内触发 OOM 。

可见这些设定值可以当作 per-cgroup 的内存分配水位线，作用于某一部分进程而非整个系统。 针对交换区使用情况也可设置这些阈值：

- **memory.swap.high** : 高交换区水位，交换区用量高于此值后会对交换区分配节流。
- **memory.swap.max** : 最大交换区限制，交换区用量高于此值后不再会发生匿名页交换。

到达这些 cgroup 设定阈值的时候，还可以设置内核回调的处理程序，从用户空间做一些程序相关的操作。

Linux 有了 cgroup v2 之后，就可以通过对某些程序设置内存用量限制，避免他们产生的页面请求把别的 程序所需的页面挤出物理内存。使用 systemd 的系统中，首先需要 启用 cgroup v2 ，在内核引导参数中加上 `systemd.unified_cgroup_hierarchy=1` 。然后开启用户权限代理：

```
1 # systemctl edit user@1000.service
2 [Service]
3 Delegate=yes
```

然后可以定义用户会话的 slice （slice 是 systemd 术语，用来映射 cgroup ），比如创建一个叫 `limit-mem` 的 slice ：

```
1 $ cat ~/.config/systemd/user/limit-mem.slice
2 [Slice]
3 MemoryHigh=3G
4 MemoryMax=4G
5 MemorySwapMax=2G
```

然后可以用 systemd-run 限制在某个 slice 中打开一个 shell：

```
$ systemd-run --user --slice=limit-mem.slice --shell
```

或者定义一个 shell alias 用来限制任意命令：

```
$ type limit-mem
limit-mem is an alias for /usr/bin/time systemd-run --user --pty --same-dir --wait --collect --slice=limit-mem.slice
$ limit-mem cp some-large-file dest/
```

实际用法有很多，可以参考 systemd 文档 man systemd.resource-control， xuanwo 也 有篇博客介绍过 systemd 下资源限制， lilydjwg 也 写过用 cgroup 限制进程内存的用法 和 用 cgroup 之后对 CPU 调度的影响。

未来展望

最近新版的 gnome 和 KDE 已经开始为桌面环境下用户程序的进程创建 systemd scope 了， 可以通过 `systemd-cgls` 观察到，每个通过桌面文件（.desktop）开启的用户空间程序 都有个独立的名字叫 `app-APPNAME-HASH.scope` 之类的 systemd scope。

有了这些 scope 之后，事实上用户程序的资源分配某种程度上已经相互独立， 不过默认的用户程序没有施加多少限制。

今后可以展望，桌面环境可以提供用户友好的方式对这些桌面程序施加公平性的限制。 不光是内存分配的大小限制，包括 CPU 和 IO 占用方面也会更公平。 值得一提的是传统的 ext4/xfs/f2fs 之类的文件系统虽然支持 cgroup writeback 节流 但是因为他们有额外的 journaling 写入，难以单独针对某些 cgroup 限制 IO 写入带宽（对文件系统元数据的写入难以统计到具体某组进程）。 而 btrfs 通过 CoW 避免了 journaling ， 在这方面有更好的支持 。相信不远的将来，复制大文件之类常见普通操作不再需要手动调用加以限制， 就能避免单个程序占用太多资源影响别的程序。