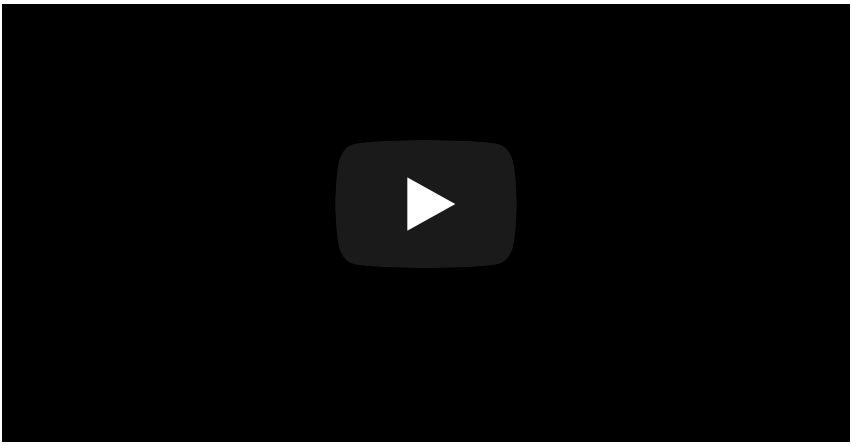
關於 swap 的一些補充

上週翻譯完 【譯】替 swap 辯護：常見的誤解 之後很多朋友們似乎還有些疑問和誤解，於是寫篇後續澄清一下。事先聲明我不是內核開發者， 這裏說的只是我的理解， 基於內核文檔中關於物理內存的描述 ，新的內核代碼的具體行爲可能和我的理解有所出入，歡迎踊躍討論。

Introduction to Memory Management in Linux



誤解1: swap 是虛擬內存，虛擬內存肯定比物理內存慢嘛

這種誤解進一步的結論通常是：「使用虛擬內存肯定會減慢系統運行時性能，如果物理內存足夠爲什麼還要用虛擬的？」 這種誤解是把虛擬內存和交換區的實現方式類比於「虛擬磁盤」或者「虛擬機」等同的方式，也隱含「先用物理內存，用完了之後用虛擬內存」也即下面的「誤解3」的理解。

首先，交換區（swap） **不是** 虛擬內存。操作系統中說「物理內存」還是「虛擬內存」的時候在指程序代碼 尋址時使用的內存地址方式，使用物理地址空間時是

在訪問物理內存，使用虛擬地址空間時是在訪問虛擬內存。 現代操作系統在大部分情況下都在使用虛擬地址空間尋址， **包括** 在執行內核代碼的時候。

並且，交換區 **不是** 實現虛擬內存的方式。操作系統使用內存管理單元（MMU，Memory Management Unit）做虛擬內存地址到物理內存地址的地址翻譯，現代架構下 MMU 通常是 CPU 的一部分，配有它專用的一小塊存儲區叫做地址轉換旁路緩存（TLB，Translation Lookaside Buffer）， 只有在 TLB 中沒有相關地址翻譯信息的時候 MMU 纔會以缺頁中斷的形式調用操作系統內核幫忙。 除了 TLB 信息不足的時候，大部分情況下使用虛擬內存都是硬件直接實現的地址翻譯，沒有軟件模擬開銷。 實現虛擬內存不需要用到交換區，交換區只是操作系統實現虛擬內存後能提供的一個附加功能， 即便沒有交換區，操作系統大部分時候也在用虛擬內存，包括在大部分內核代碼中。

誤解2: 但是沒有交換區的話，虛擬內存地址都有物理內存對應嘛

很多朋友也理解上述操作系統實現虛擬內存的方式，但是仍然會有疑問：「我知道虛擬內存和交換區的區別， 但是沒有交換區的話，虛擬內存地址都有物理內存對應，不用交換區的話就不會遇到讀虛擬內存需要讀寫磁盤 導致的卡頓了嘛」。

這種理解也是錯的，禁用交換區的時候，也會有一部分分配給程序的虛擬內存不對應物理內存， 比如使用 mmap 調用實現內存映射文件的時候。實際上即便是使用 read/write 讀寫文件， Linux 內核中（可能現代操作系統內核都）在底下是用和 mmap 相同的機制建立文件 到虛擬地址空間的地址映射，然後實際讀寫到虛擬地址時靠缺頁中斷把文件內容載入頁面緩存（page cache）。內核加載可執行程序和動態鏈接庫的方式也是通過內存映射文件。甚至可以進一步說， 用戶空間的虛擬內存地址範圍內，除了匿名頁之外，其它虛擬地址都是文件後備（backed by file），而匿名頁通過交換區作爲文件後備。上篇文章中提到的別的類型的內存，比如共享內存頁面（shm）是被一個內存中的虛擬文件系統後備的，這一點有些套娃先暫且不提。於是事實是無論有沒有交換區， 缺頁的時候總會有磁盤讀寫從慢速存儲加載到物理內存，這進一步引出上篇文章中對於交換區和頁面緩存這兩者的討論。

誤解3： 不是內存快用完的時候才會交換的麼？

簡短的答案可以說「是」，但是內核理解的「內存快用完」和你理解的很可能不同。 也可以說「不是」，就算按照內核理解的「內存快用完」的定義，內存快用完的時候內核的行爲是去回收內存， 至於回收內存的時候內核會做什麼有個複雜的啓發式經驗算法，實際上真的內存快滿的時候根本來不及做 swap ，內核可能會嘗試丟棄 page cache 甚至丟棄 vfs cache (dentry cache / inode cache) 這些不需要磁盤I/O就能更快獲取可用內存的動作。

深究這些內核機制之前，我在思考爲什麼很多朋友會問出這樣的問題。可能大部分這麼問的人，學過編程， 稍微學過基本的操作系統原理，在腦海裏對內核分配頁面留着這樣一種印象（C僞代碼）：

```
1  ////////////////////  userspace space  ////////////////
2  void* malloc(int size){
3      void* pages = mmap(...);                    // 從內核分配內存頁
4      return alloc_from_page(pages, size);        // 從拿到的內存頁細分
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      // 可用內存不夠了！嘗試搞點內存
16      page_frame_info* least_accessed =
 lru_pop_page_frame();     // 找出最少訪
問的頁面
17      switch ( least_accessed -> pf_ty
pe ){
18        case PAGE_CACHE: drop_page_cac
he(least_accessed); break;  // 丟棄文件緩存

19        case SWAP:       swap_out(leas
t_accessed);        break;  // <- 寫磁盤
，所以系統卡了！
20        // ... 別的方式回收 least_access
ed
21      }
22      append_free_page(free_page_list,
least_accessed);           // 回收到的頁
面加入可用列表
23      available_mem += least_accessed
-> size;
24    }
25    // 搞到內存了！返回給程序
```

```
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  }
```

這種邏輯隱含三層 **錯誤的** 假設：

1. 分配物理內存是發生在從內核分配內存的時候的，比如 `malloc/mmap` 的時候。
2. 內存回收是發生在進程請求內存分配的上下文裏的，換句話說進程在等內核的內存回收返回內存，不回收到內存，進程就得不到內存。
3. 交換出內存到 swap 是發生在內存回收的時候的，會阻塞內核的內存回收，進而阻塞程序的內存分配。

這種把內核代碼當作「具有特權的庫函數調用」的看法，可能很易於理解， 甚至早期可能的確有操作系統的內核是這麼實現的，但是很可惜現代操作系統都不是這麼做的。 上面三層假設的錯誤之處在於：

1. 在程序請求內存的時候，比如 `malloc/mmap` 的時候，內核只做虛擬地址分配， 記錄下某段虛擬地址空間對這個程序是可以合法訪問的，但是不實際分配物理內存給程序。 在程序第一次訪問到虛擬地址的時候，才會實際分配物理內存。這種叫 **惰性分配（lazy allocation）** 。

2. 在內核感受到內存分配壓力之後，早在內核內存用盡之前，內核就會在後臺慢慢掃描並回收內存頁。內存回收通常不發生在內存分配的時候，除非在內存非常短缺的情況下，後臺內存回收來不及滿足當前分配請求， 纔會發生 **直接回收(direct reclamation)** 。
3. 同樣除了直接回收的情況，大部分正常情況下換出頁面是內存管理子系統調用 DMA 在後臺慢慢做的， 交換頁面出去不會阻塞內核的內存回收，更不會阻塞程序做內存分配（malloc ）和使用內存(實際訪問惰性分配的內存頁)。

也就是說，現代操作系統內核是高度並行化的設計，內存分配方方面面需要消耗計算資源或者 I/O 帶寬的場景，都會儘量並行化，最大程度利用好計算機所有組件（CPU/MMU/DMA/IO）的吞吐率， 不到緊要關頭需要直接回收的場合，就不會阻塞程序的正常執行流程。

惰性分配有什麼好處？

或許會有人問：「我讓你分配內存，你給我分配了個虛擬的，到用的時候還要做很多事情才能給我，這不是騙人嘛」， 或者會有人擔心惰性分配會對性能造成負面影響。

這裏實際情況是程序從分配虛擬內存的時候，「到用的時候」，這之間有段時間間隔，可以留給內核做準備。程序可能一下子分配一大片內存地址，然後再在執行過程中解析數據慢慢往地址範圍內寫東西。程序分配虛擬內存的速率可以是「突發」的，比如一個系統調用中分配 1GiB 大小，而實際寫入數據的速率會被 CPU 執行速度等因素限制，不會短期內突然寫入很多頁面。這個分配速率導致的時間差內內核可以完成很多後臺工作，比如回收內存，比如把回收到的別的進程用過的內存頁面初始化爲全0，這部分後臺工作可以和程序的執行過程並行，從而當程序實際用到內存的時候，需要的準備工作已經做完了，大部分場景下可以直接分配物理內存出來。

如果程序要做實時響應，想避免因爲惰性分配造成的性能不穩定，可以使用 `mlock/mlockall` 將得到的虛擬內存鎖定在物理內存中，鎖的過程中內核會做物理內存分配。不過要區分「性能不穩定」和「低性能」，預先分配內存可以避免實際使用內存時分配物理頁面的額外開銷，但是會拖慢整體吞吐率，所以要謹慎使用。

很多程序分配了很大一片地址空間，但是實際並不會用完這些地址，直到程序執行結束這些虛擬地址也一直 處於沒有對應物理地址的情況。惰性分配可以避免爲這些情況浪費物理內存頁面，使得很多程序可以無憂無慮地 隨意分配內存地址而不用擔心性能損失。這種分配方式也叫「超額分配（overcommit）」。飛機票有超

售， VPS 提供商劃分虛擬機有超售，操作系統管理內存也同樣有這種現象，合理使用超額分配能改善整體系統效率。

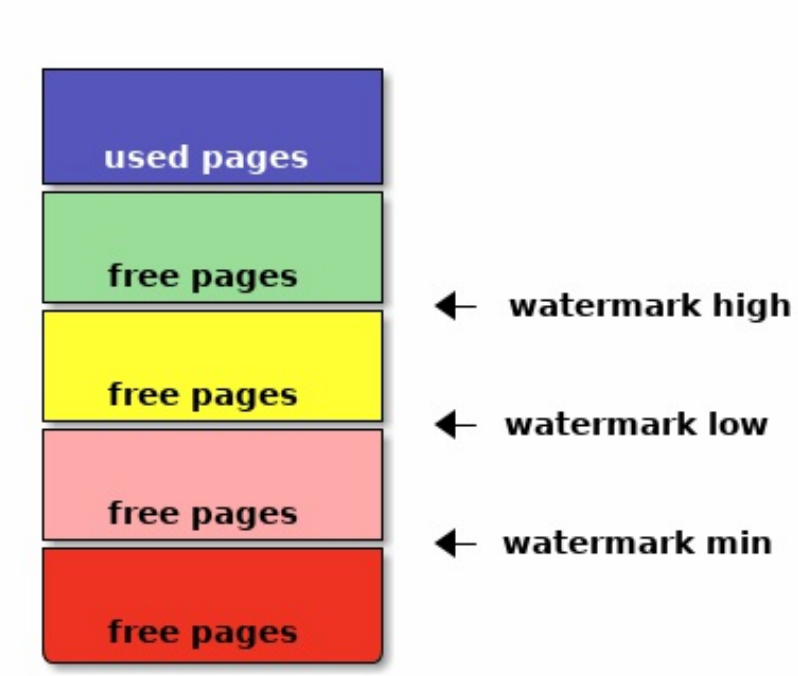
內核要高效地做到惰性分配而不影響程序執行效率的前提之一，在於程序真的用到內存的時候， 內核能不做太多操作就立刻分配出來，也就是說內核需要時時刻刻在手上留有一部分空頁， 滿足程序執行時內存分配的需要。換句話說，內核需要早在物理內存用盡之前，就開始回收內存。

那麼內核什麼時候會開始回收內存？

首先一些背景知識：物理內存地址空間並不是都平等，因爲一些地址範圍可以做 DMA 而另一些不能，以及 NUMA 等硬件環境傾向於讓 CPU 訪問其所在 NUMA 節點內存範圍。在 32bit 系統上內核的虛擬地址空間還有低端內存和高端內存的區分，他們會傾向於使用不同屬性的物理內存，到 64bit 系統上已經沒有了這種限制。

硬件限制了內存分配的自由度，於是內核把物理內存空間分成多個 Zone ，每個 Zone 內各自管理可用內存， Zone 內的內存頁之間是相互平等的。

zone 內水位線



一個 Zone 內的頁面分配情況可以右圖描繪。 除了已用內存頁，剩下的就是空閒頁（free pages），空閒頁範圍中有三個水位線（watermark）評估當前內存壓力情況，分別是高位（high）、低位（low）、最小位（min）。

當內存分配使得空閒頁水位低於低位線，內核會喚醒 kswapd 後臺線程， kswapd 負責掃描物理頁面的使用情況並挑選一部分頁面做回收，直到可用頁面數量恢復到水位線高位（high）以上。 如果 kswapd 回收內存

的速度慢於程序執行實際分配內存的速度， 可用空閒頁數量可能進一步下降，降至低於最小水位（min）之後，內核會讓內存分配進入 **直接回收(direct reclamation)** 模式，在直接回收模式下，程序分配某個物理頁的請求（ 第一次訪問某個已分配虛擬頁面的時候）會導致在進程上下文中阻塞式地調用內存回收代碼。

除了內核在後臺回收內存，進程也可以主動釋放內存，比如有程序退出的時候就會釋放一大片內存頁， 所以可用頁面數量可能會升至水位線高位以上。有太多可用頁面浪費資源對整體系統運行效率也不是好事， 所以系統會積極緩存文件讀寫，所有 page cache 都留在內存中，直到可用頁面降至低水位以下觸發 kswapd 開始工作。

設置最小水位線（min）的原因在於，內核中有些硬件也會突然請求大量內存，比如來自網卡接收到的數據包， 預留出最小水位線以下的內存給內核內部和硬件使用。

設置高低兩個控制 kswapd 開關的水位線是基於控制理論。喚醒 kswapd 掃描內存頁面本身有一定計算開銷，於是每次喚醒它幹活的話就讓它多做一些活（ high - low ），避免頻繁多次喚醒。

因爲有這些水位線，系統中根據程序請求內存的「速率」，整個系統的內存分配在宏觀的一段時間內可能處於以下幾種狀態：

1. **不回收：** 系統中的程序申請內存速度很慢，或者程序主動釋放內存的速度很快， （比如程序執行

時間很短，不怎麼進行文件讀寫就馬上退出，）此時可用頁面數量可能一直處於低水位線以上， 內核不會主動回收內存，所有文件讀寫都會以頁面緩存的形式留在物理內存中。

2. **後臺回收：** 系統中的程序在緩慢申請內存，比如做文件讀寫， 比如分配並使用匿名頁面。系統會時不時地喚醒 `kswapd` 在後臺做內存回收， 不會干擾到程序的執行效率。
3. **直接回收：** 如果程序申請內存的速度快於 `kswapd` 後臺回收內存的速度， 空閒內存最終會跌破最小水位線，隨後的內存申請會進入直接回收的代碼路徑，從而極大限制內存分配速度。 在直接分配和後臺回收的同時作用下，空閒內存可能會時不時回到最小水位線以上， 但是如果程序繼續申請內存，空閒內存量就會在最小水位線附近上下徘徊。
4. **殺進程回收：** 甚至直接分配和後臺回收的同時作用也不足以拖慢程序分配內存的速度的時候， 最終空閒內存會完全用完，此時觸發 OOM 殺手幹活殺進程。

系統狀態處於 **1. 不回收** 的時候表明分配給系統的內存量過多，比如系統剛剛啓動之類的時候。 理想上應該讓系統長期處於 **2. 後臺回收** 的狀態，此時最大化利用緩存的效率而又不會因爲內存回收 減緩程序執行速度。如果系統引導後長期處於 **1. 不回收** 的狀態下，那麼說明沒有充分利用空閒內存做 文件緩存，有些 unix 服務比如 preload 可用來提前填充文件緩存。

如果系統頻繁進入 **3. 直接回收** 的狀態，表明在這種工作負載下系統需要減慢一些內存分配速度， 讓 kswapd 有足夠時間回收內存。就如前一篇翻譯中 Chris 所述，頻繁進入這種狀態也不一定代表「內存不足」，可能表示內存分配處於非常高效的利用狀態下， 系統充分利用慢速的磁盤帶寬，爲快速的內存緩存提供足夠的可用空間。 **直接回收** 是否對進程負載有負面影響要看具體負載的特性。 此時選擇禁用 swap 並不能降低磁盤 I/O，反而可能縮短 **2. 後臺回收** 狀態能持續的時間， 導致更快進入 **4. 殺進程回收** 的極端狀態。

當然如果系統長期處於 **直接回收** 的狀態的話，則說明內存總量不足，需要考慮增加物理內存， 或者減少系統負載了。如果系統進入 **4. 殺進程回收** 的狀態，不光用空間的進程會受影響， 並且還可能導致內核態的內存分配受影響，產生網絡丟包之類的結果。

微調內存管理水位線

可以看一下運行中的系統中每個 Zone 的水位線在哪兒。比如我手上這個 16GiB 的系統中：

```
$ 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
```

因爲不是 NUMA 系統，所以只有一個 NUMA node，其中根據 DMA 類型共有 3 個 Zone 分別叫 DMA, DMA32, Normal 。三個 Zone 的物理地址範圍（spanned）加起來大概有 \(4095+1044480+3407872\)

大約 17GiB 的地址空間，而實際可訪問的地址範圍（present）加起來有 \(3997+780044+3407872\) 大約 16GiB 的可訪問物理內存。

其中空閒頁面有 \(3459+762569+1460218\) 大約 8.5GiB，三條水位線分別在：\(\texttt{high} = 24+4710+20607 = 98\texttt{MiB}\)，\(\texttt{low} = 20+3925+17173 = 82\texttt{MiB}\)，\(\texttt{min} = 16+3140+13739 = 65\texttt{MiB}\) 的位置。

具體這些水位線的確定方式基於幾個 sysctl。首先 min 基於 `vm.min_free_kbytes` 默認是基於內核低端內存量的平方根算的值，並限制到最大 64MiB 再加點餘量，比如我這臺機器上 `vm.min_free_kbytes = 67584`，於是 min 水位線在這個位置。其它兩個水位線基於這個計算，在 min 基礎上增加總內存量的 `vm.watermark_scale_factor / 10000` 比例（在小內存的系統上還有額外考慮），默認 `vm.watermark_scale_factor = 10` 在大內存系統上意味着 low 比 min 高 0.1%，high 比 low 高 0.1%。

可以手動設置這些值，以更早觸發內存回收，比如將 `vm.watermark_scale_factor` 設爲 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
```

得到的三條水位線分別在 $\texttt{min} = 16+3149+13729 = 66\texttt{MiB}$ ， $\texttt{low} = 55+10785+47013 = 226\texttt{MiB}$ ， $\texttt{high} = 94+18421+80297 = 386\texttt{MiB}$ ， 從而 low 和 high 分別比 min 提高 160MiB 也就是內存總量的 1% 左右。

在 swap 放在 HDD 的系統中，因爲換頁出去的速度較慢，除了上篇文章說的降低 `vm.swappiness` 之外，還可以適當提高 `vm.watermark_scale_factor` 讓內核更早開始回收內存，這雖然會稍微降低緩存命中率，但是另一方面可以在進入直接回收模式之前 有更多時間做後臺換頁，也將有助於改善系統整體流暢度。

只有 0.1%，這不就是說內存快用完的時候麼？

所以之前的「誤解3」我說答案可以說「是」或者「不是」，但是無論回答是或不是，都代表了認爲「swap 就是額外的慢速內存」的錯誤看法。當有人在強調「swap 是內存快用完的時候才交換」的時候， 隱含地，是在把系統總體的內存分配看作是一個靜態的劃分過程：打個比方這就像在說，我的系統裏存儲空間有快速 128GiB SSD 和慢速 HDD 的 1TiB ，同樣內存有快速的 16GiB RAM 和慢速 16GiB 的 swap 。 這種靜態劃分的

類比是錯誤的看待方式，因爲系統回收內存進而做頁面交換的方式是動態平衡的過程， 需要考慮到「時間」和「速率」而非單純看「容量」。

假設 swap 所在的存儲設備可以支持 5MiB/s 的吞吐率（ HDD 上可能更慢， SSD 上可能更快，這裏需要關注數量級），相比之下 DDR3 大概有 10GiB/s 的吞吐率，DDR4 大概有 20GiB/s ，無論多快的 SSD 也遠達不到這樣的吞吐（可能 Intel Optane 這樣的 DAX 設備會改變這裏的狀況）。從而把 swap 當作慢速內存的視角來看的話，加權平均的速率是非常悲觀的，「 16G 的 DDR3 + 16G 的 swap 會有 $\frac{16 \times 10 \times 1024 + 16 \times 5}{16+16} = 5 \texttt{GiB/s}$ 的吞吐？所以開 swap 導致系統速度降了一半？」顯然不能這樣看待。

動態的看待方式是， swap 設備能提供 5MiB/s 的吞吐，這意味着：如果我們能把未來 10 分鐘內不會訪問到的頁面換出到 swap ，那麼就相當於有 $10 \times 60 \texttt{s} \times 5 \texttt{MiB/s} = 3000 \texttt{MiB}$ 的額外內存，用來放那 10 分鐘內可能會訪問到的頁面緩存。 10 分鐘只是隨口說的一段時間，可以換成 10 秒或者 10 小時，重要的是只要頁面交換發生在後臺， 不阻塞前臺程序的執行，那麼 swap 設備提供的額外吞吐率相當於一段時間內提供了更大的物理內存， 總是能提升頁面緩存的命中，從而改善系統性能。

當然系統內核不能預知「未來 10 分鐘內需要的頁面」，只能根據歷史上訪問內存的情況預估之後可能會訪問的情況， 估算不準的情況下，比如最近10分鐘內用過的頁面緩存在之後10分鐘內不再被使用的時候， 爲了

把最近這10分鐘內訪問過的頁面留在物理內存中，可能會把之後10分鐘內要用到的匿名頁面換出到了交換設備上。 於是會有下面的情況：

但是我開了 swap 之後，一旦複製大文件，系統就變卡，不開 swap 不會這樣的

大概電腦用戶都經歷過這種現象，不限於 Linux 用戶，包括 macOS 和 Windows 上也是。 在文件管理器中複製了幾個大文件之後，切換到別的程序系統就極其卡頓，複製已經結束之後的一段時間也會如此。 複製的過程中系統交換區的使用率在上漲，複製結束後下降，顯然 swap 在其中有重要因素，並且禁用 swap 或者調低 swappiness 之後就不會這樣了。於是網上大量流傳着解釋這一現象，並進一步建議禁用 swap 或者調低 swappiness 的文章。我相信不少關心系統性能調優的人看過這篇「Tales from responsivenessland: why Linux feels slow, and how to fix that」或是它的轉載、翻譯，用中文搜索的話還能找到更多錯誤解釋 swappiness 目的的文章，比如 這篇將 swappiness 解釋成是控制內存和交換區比例的參數。

除去那些有技術上謬誤的文章，這些網文中描述的現象是有道理的，不單純是以訛傳訛。 桌面環境中內存分配策略的不確定性和服務器環境中很不一樣，複製、下載、解壓大文件等導致一段時間內 大量佔用頁面緩存，以至於把操作結束後需要的頁面攆出物理內存，無論是交換出去的方式還是以丟棄頁面緩存的方式， 都會導致桌面響應性降低。

不過就像前文 Chris 所述，這種現象其實並不能通過禁止 swap 的方式緩解：禁止 swap 或者調整 swappiness 讓系統儘量避免 swap 只影響回收匿名頁面的策略，不影響系統回收頁面的時機， 也不能避免系統丟棄將要使用的頁面緩存而導致的卡頓。

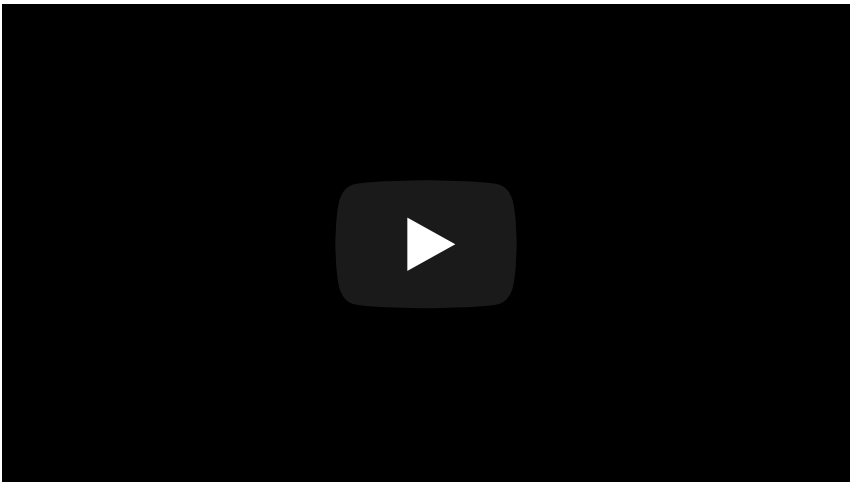
以前在 Linux 上也沒有什麼好方法能避免這種現象。 macOS 轉用 APFS 作爲默認文件系統之後， 從文件管理器（Finder）複製文件默認啓用 file clone 快速完成，這操作不實際複製文件數據， 一個隱含優勢在不需要讀入文件內容，從而不會導致大量頁面緩存失效。 Linux 上同樣可以用支持 reflink 的文件系統比如 btrfs 或者開了 reflink=1 的 xfs 達到類似的效果。 不過 reflink 也只能拯救複製文件的情況，不能改善解壓文件、下載文件、計算文件校驗等情況下， 一次性處理大文件對內存產生的壓力。

好在最近幾年 Linux 有了 cgroup ，允許更細粒度地調整系統資源分配。進一步現在我們有了 cgroup v2 ，前面 Chris 的文章也有提到 cgroup v2 的 `memory.low` 可以某種程度上建議內存子系統 儘量避免回收某些 cgroup 進程的內存。

於是有了 cgroup 之後，另一種思路是把複製文件等大量使用內存而之後又不需要保留頁面緩存的程序單獨放入 cgroup 內限制它的內存用量，用一點點複製文件時的性能損失換來整體系統的響應流暢度。

關於 cgroup v1 和 v2

稍微跑題說一下 cgroup v2 相對於 v1 帶來的優勢。這方面優勢在 Chris Down 另一個關於 cgroup v2 演講 中有提到。老 cgroup v1 按控制器區分 cgroup 層級，從而內存控制器所限制的東西和 IO 控制器所限制的東西是獨立的。在內核角度來看，頁面寫回（page writeback）和交換（swap）正是夾在內存控制器和IO控制器管理的邊界上，從而用 v1 的 cgroup 難以同時管理。 v2 通過統一控制器層級解決了這方面限制。具體見下面 Chris Down 的演講。



用 cgroup v2 限制進程的內存分配

實際上有了 cgroup v2 之後，還有更多控制內存分配的方案。 cgroup v2 的內存控制器 可以對某個 cgroup 設置這些閾值：

- **memory.min** : 最小內存限制。內存用量低於此值後系統不會回收內存。
- **memory.low** : 低內存水位。內存用量低於此值後系統會儘量避免回收內存。
- **memory.high** : 高內存水位。內存用量高於此值後系統會積極回收內存，並且會對內存分配節流（throttle）。
- **memory.max** : 最大內存限制。內存用量高於此值後系統會對內存分配請求返回 ENOMEM，或者在 cgroup 內觸發 OOM 。

可見這些設定值可以當作 per-cgroup 的內存分配水位線，作用於某一部分進程而非整個系統。 針對交換區使用情況也可設置這些閾值：

- **memory.swap.high** : 高交換區水位，交換區用量高於此值後會對交換區分配節流。
- **memory.swap.max** : 最大交換區限制，交換區用量高於此值後不再會發生匿名頁交換。

到達這些 cgroup 設定閾值的時候，還可以設置內核回調的處理程序，從用戶空間做一些程序相關的操作。

Linux 有了 cgroup v2 之後，就可以通過對某些程序設置內存用量限制，避免他們產生的頁面請求把別的 程序所需的頁面擠出物理內存。使用 systemd 的系統中，首先需要 啓用 cgroup v2 ，在內核引導參數中加上 `systemd.unified_cgroup_hierarchy=1` 。然後開啓用戶權限代理：

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

然後可以定義用戶會話的 slice （slice 是 systemd 術語，用來映射 cgroup ），比如創建一個叫 `limit-mem` 的 slice ：

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

然後可以用 systemd-run 限制在某個 slice 中打開一個 shell：

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

或者定義一個 shell alias 用來限制任意命令：

```
$ 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/
```

實際用法有很多，可以參考 systemd 文檔 man systemd.resource-control， xuanwo 也有篇博客介紹過 systemd 下資源限制， lilydjwg 也寫過用 cgroup 限制進程內存的用法和用 cgroup 之後對 CPU 調度的影響。

未來展望

最近新版的 gnome 和 KDE 已經開始爲桌面環境下用戶程序的進程創建 systemd scope 了， 可以通過 `systemd-cgls` 觀察到，每個通過桌面文件（.desktop）開啓的用戶空間程序 都有個獨立的名字叫 `app-APPNAME-HASH.scope` 之類的 systemd scope。

有了這些 scope 之後，事實上用戶程序的資源分配某種程度上已經相互獨立， 不過默認的用戶程序沒有施加多少限制。

今後可以展望，桌面環境可以提供用戶友好的方式對這些桌面程序施加公平性的限制。 不光是內存分配的大小限制，包括 CPU 和 IO 佔用方面也會更公平。 值得一提的是傳統的 ext4/xfs/f2fs 之類的文件系統雖然支持 cgroup writeback 節流 但是因爲他們有額外的 journaling 寫入，難以單獨針對某些 cgroup 限制 IO 寫入帶寬（對文件系統元數據的寫入難以統計到具體某組進程）。 而 btrfs 通過 CoW 避免了 journaling， 在這方面有更好的支持 。相信不遠的將來，複製大文件之類常見普通操作不再需要手動調用加以限制， 就能避免單個程序佔用太多資源影響別的程序。