zhuanlan.zhihu.com /p/352502252

[小长文]Linux 内存管理

大纲

- 进程的地址空间布局
- 从虚拟内存到物理内存
  - 虚拟内存
  - 页表的设计
  - 缺页中断
  - TLB
  - Hugepages
  - THP
- Page Cache
  - free 命令
  - drop caches
  - 刷新脏页
  - Page Cache 相关工具
- 内存回收和内存交换
  - Swap 分区
  - 内存水位标记
  - 内存回收和 swappiness
- 总结
- 参考资料

进程的地址空间布局

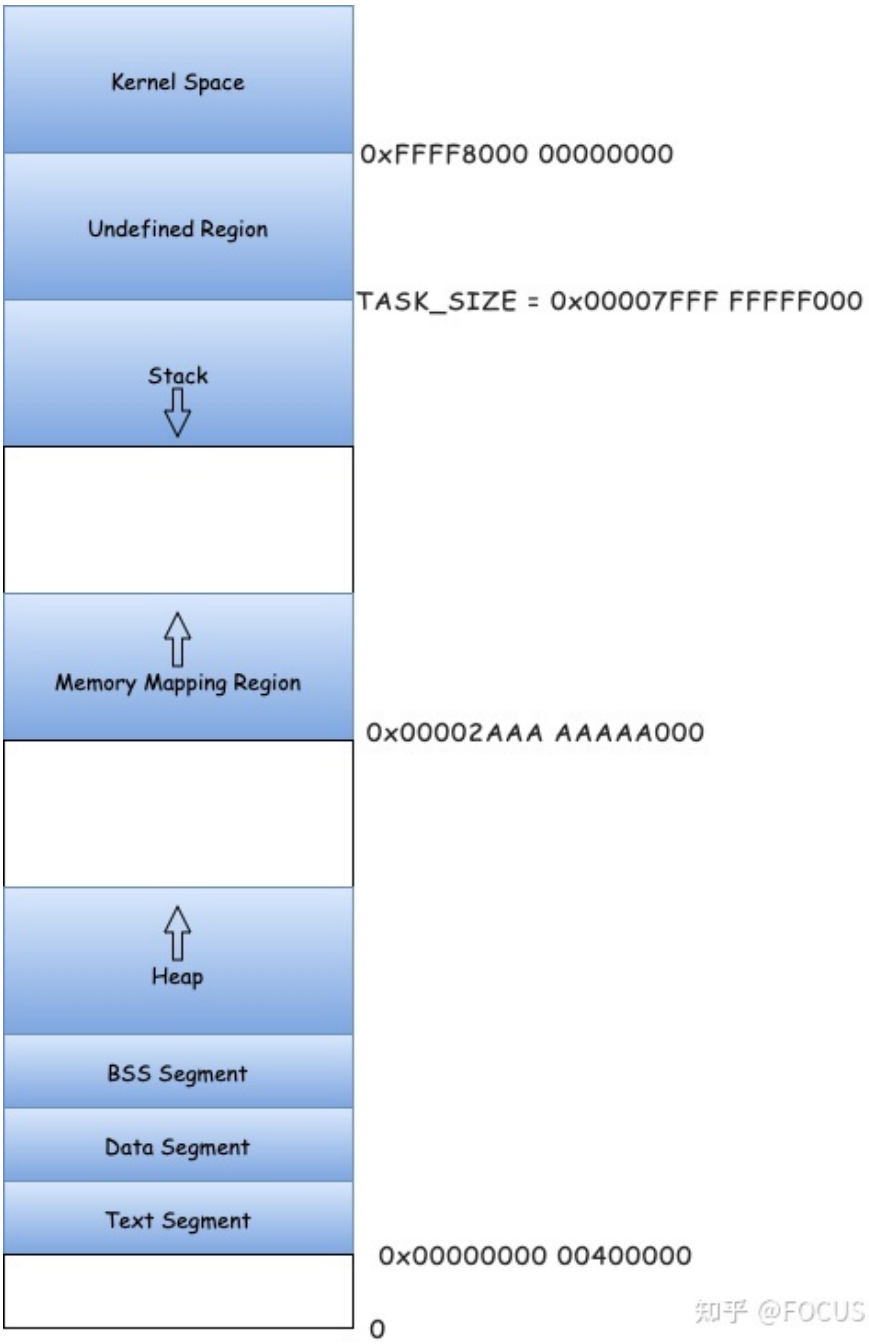
在 Linux 下，每个进程都拥有独立的虚拟地址空间。

在 IA-32 的场景下，虚拟地址只有 32 位，所以最大的寻址空间是 2^32 = 4GB。Linux 内核将这个 4GB 的地址空间按照 3:1 的比例划分，其中用户空间占用低地址的 3GB，内核空间占用高地址的 1GB。4GB 的地址空间，真的是捉襟见肘。为此，内核做了不少复杂的虚拟地址到物理地址的映射关系。不过，现在的生产环境中基本都看不到 32 位 CPU 的影子，这里就不对 32 位的地址空间做深入研究了。

在 AMD-64 的场景下，其虚拟地址是 48 位（不是 64 位），整个地址空间足足有 2^48 = 256TB 这么大。Linux 内核将其按照 1:1 的比例划分。同时，为了保留扩展到 64 位地址空间的能力，Linux 将 64 位的地址区间划分了三个部分：

1. 用户地址空间，范围是 0x00000000 00000000 ~ 0x00007FFF FFFFFFFF，大小是 128TB。
2. 内核地址空间，范围是 0xFFFF8000 00000000 ~ 0xFFFFFFFF FFFFFFFF，大小也是 128TB。
3. 用户空间和内核空间之间留下了一个 undefined region，用于未来支持更大的地址空间。

64 位的进程地址空间布局如下：



- Text Segment，存储程序的二进制代码。
- Data Segment，存储已初始化的全局数据。
- BSS Segment，存储未初始化的全局数据。
- Heap：堆内存，向高地址增长。
- Memory Mapping Region：mmap 系统调用映射内存的地址区间。
- Stack：栈内存，向低地址增长。
- Undefined Region：未定义的地址区间，用于将来扩展 64 位地址空间。
- Kernel Space：内核地址空间。
- **为了防止缓存区溢出攻击，堆、栈、mmap 映射区域并不是从一个固定地址开始**——程序在启动时随机改变这些值的设置，使得使用缓冲区溢出进行攻击更加困难。这个特性在 Linux 下叫做 ASLR (Address Space Layout Randomisation)，可以通过设置 /proc/sys/kernel/randomize_va_space 来控制 ASLR，有 0、1 和 2 三种选项：
  - 0: 禁用 ASLR。
  - 1：加载（mmap）共享库使用 ASLR。

- 2：加载（mmap）共享库和 brk 的起始地址使用 ASLR。

程序可以通过调用 `brk` / `sbrk` 来修改 heap 的结束地址。

```
int brk(void *addr);
void *sbrk(intptr_t increment);
```

- brk - 将堆的结束地址设置为 addr。
- sbrk - 将堆大小增加 increment 字节，并返回增加 increment 字节之前的堆结束地址。

mmap 的作用是在 Memory Mapping Region 建立一个内存与文件、设备等的映射，也可以建立匿名映射（共享内存）。

```
void* mmap(void* start, size_t length, int prot, int flags, int fd, off_t offset);
```

下面，我们用一个简单的例子来探究一下进程的地址空间。

```
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <unistd.h>

int data = 0;

int main() {
  static int bss;
  int stack;

  printf("Memory Layout of Process: %d\n", getpid());
  printf("Text Address %p\n", main);
  printf("Data Address %p\n", &data);
  printf("BSS Address %p\n", &bss);
  printf("Stack Address %p\n", &stack);
  void* heap_start = sbrk(4096);
  printf("Heap Start Address: %p\n", heap_start);
  void* heap_end = sbrk(0);
  printf("Heap End Address(Head Start +4096): %p\n", heap_end);
  printf("Allocate Memory Size %ld\n", (char*)heap_end - (char*)heap_start);
  void* mmap_addr = mmap(NULL, 1024, PROT_READ | PROT_WRITE,
                         MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
  printf("Memory Map Address %p\n", mmap_addr);

  return 0;
}
```

从虚拟内存到物理内存

虚拟内存

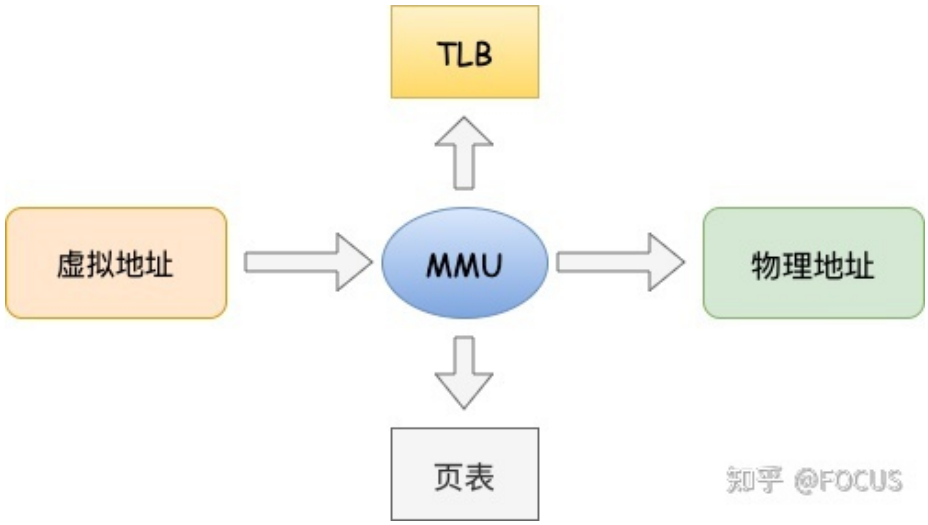
前面说的进程地址空间，其实是一个虚拟的内存地址空间——Linux 内核使用一种叫做“虚拟内存”的技术，让每个进程都认为自己使用的是一块大的连续的内存（虚拟地址空间）。

虚拟地址把不同进程隔离起来，避免相互影响。事实上，每个进程使用的内存散布在物理内存的不同区域，或者可能被 swap 到硬盘中。

一般情况下，我们所说的“内存地址”都是虚拟地址。C/C++ 代码中的指针实际上就是虚拟地址。指针解引用，其实就是 CPU 要访问一个虚拟地址，此时这个虚拟地址需要转换成物理地址。

那么虚拟地址转换成物理地址，这个过程具体怎么实现呢？简单说就是，每个进程都有一套自己的页表，虚拟地址通过查表的方式转换成物理地址：

1. 在进程切换时，CPU 会把新进程的页表地址填入 CPU 的 CR3 寄存器，供 MMU 使用。
2. 当进程有地址访问时，MMU 会根据虚拟地址在页表中找到对应的物理地址。
3. 为了提高虚拟地址到物理地址的转换性能，CPU 内部增加了一个页表的 cache，叫做 TLB。每次转换先从 TLB 查找，找不到再到页表查找。

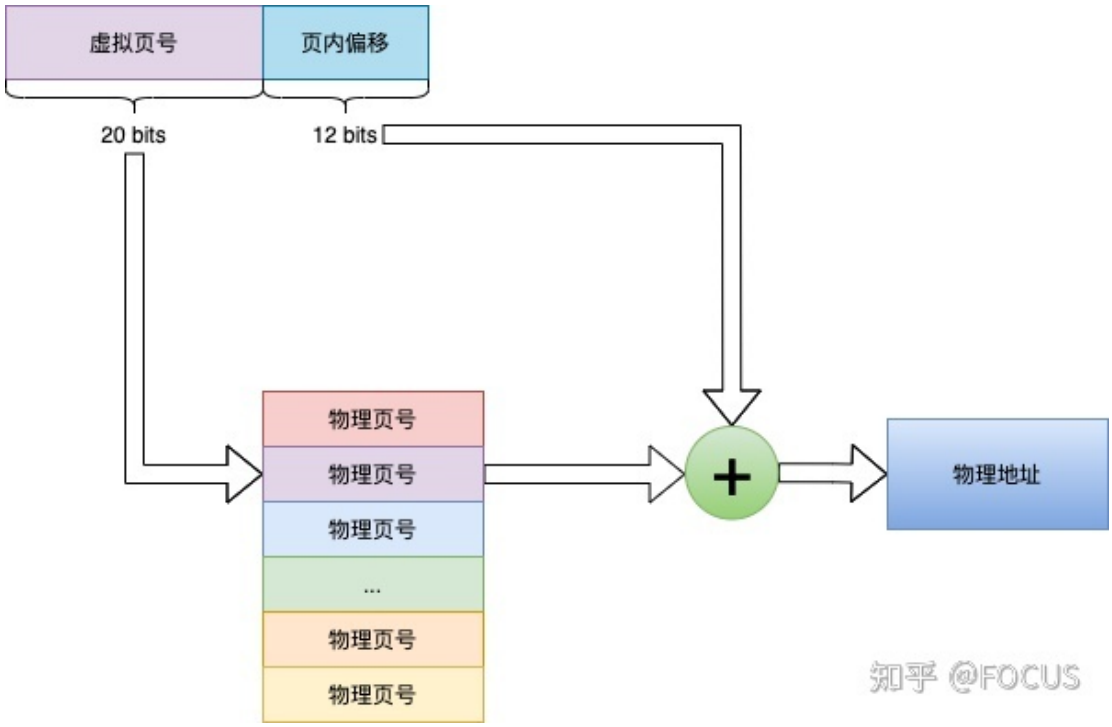


页表的设计

Linux 下，虚拟内存和物理内存都是分页管理的，一般内存页的大小是 4KB。

页表，其实就是建立一个虚拟内存页到物理内存页的映射。最简单的做法：

1. 将虚拟地址划分成虚拟页号和页内偏移两部分。
2. 用一个一维数组来建立虚拟页号到物理页号的映射，虚拟页号作为一维数组的索引下标。

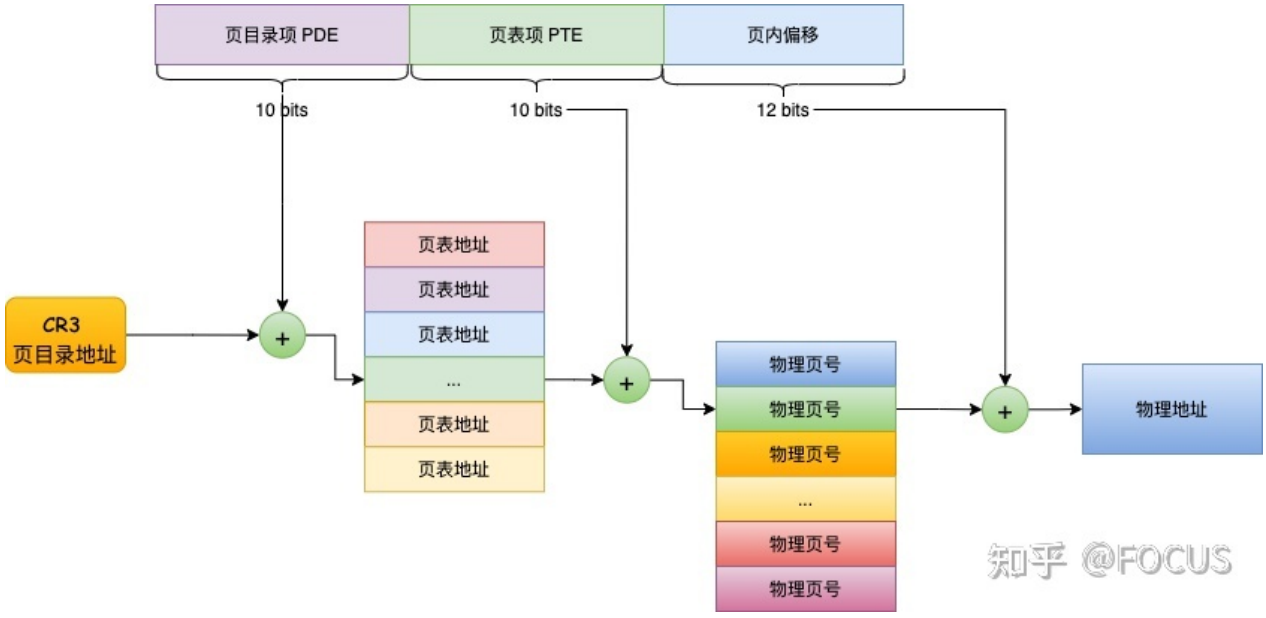


这个结构下的页表查找速度非常快，只需要一次内存的随机读取。但是，内存开销较高。以 32 位的虚拟地址为例，页内偏移部分需要 12 位（2^12B = 4KB），虚拟页号部分是 20 位。因此，这个一维数组需要 2^20 个元素，那个元素大小是 4B，占用内存 2^20 * 4B = 4MB。

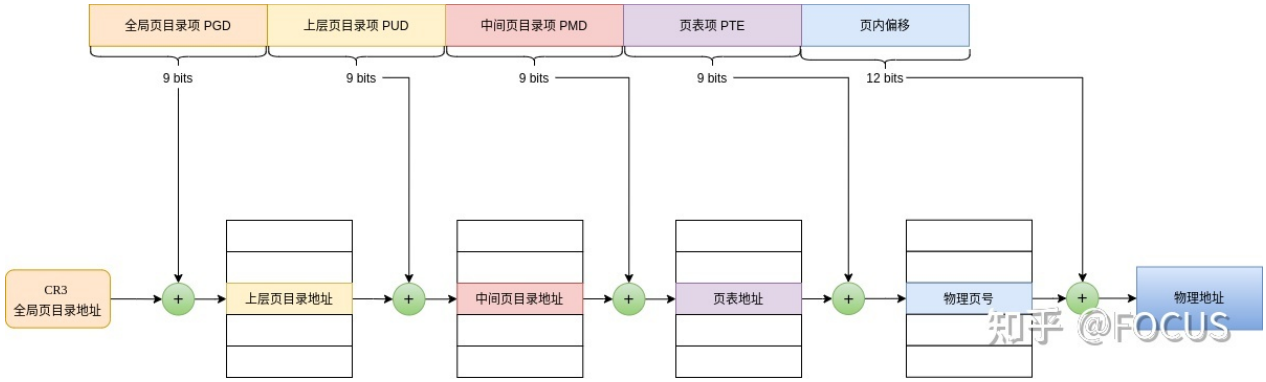
4MB 内存，看起来微不足道，但这仅仅是一个进程占用的。正常运行的时候，系统至少都会有几十、几百个进程。100 个进程，就要占用 400MB 的内存，而 32 位的系统，最大的内存也才 4GB。对于 64 位系统来说，虚拟页号部分至少是 36 位。这种页表结构的内存开销完全不可能接受。

为了解决一级页表占用的内存太多的问题，Linux 采用了一种“多级页表”的结构。所谓多级，就是把一级页表中的虚拟页号分成多段。比如：

Linux 32 位系统采用了两级页表：虚拟地址（32 bits） = 页目录项 PDE（Page Directory Entry，10 bits） + 页表项 PTE（Page Table Entry，10 bits） + 页内偏移 Offset（12 bits）。



Linux 64 位系统采用了四级页表：虚拟地址（48 bits） = 全局页目录项 PGD（Page Global Directory，9 bits）+ 上层页目录项 PUD（Page Upper Directory，9 bits） + 中间页目录项 PMD（Page Middle Directory，9 bits）+ 页表项 PTE（Page Table Entry，9 bits）+ 页内偏移 Offset（12 bits）



Linux 4.11 开始支持五级页表，虚拟地址空间从 48 bits 扩展到 57 bits。内核在 PGD 和 PUD 之间增加了一个叫 P4D 的层次。

实际上，如果全部虚拟地址都需要映射到物理地址，多级页表的开销是大于一级页表的。比如，32 位虚拟地址空间的二级页表，如果要映射全部虚拟地址，需要的内存大小为：PDE 的大小 + PTE 的大小 = 4 * 2^10 B + 4 * 2^10 * 2^10 B = 4KB + 4MB。

那为什么还说多级页表比一级页表节省内存呢？

多级页表之所以能比一级页表节省内存，是因为：**并不是所有进程的所有虚拟地址都会被用到，没有用到的虚拟地址没有必要建立虚拟地址到物理地址之间的映射关系。** 更具体的：

1. 对于 32 位地址空间，虽然只有 4GB，但是并不是所有进程都会用到 4GB，所以没有必要每个进程一开始就建立完整的地址映射关系。
2. 对于 64 位地址空间，考虑到 64 位的虚拟地址空间非常庞大——远大于实际的物理内存。一个进程更加不可能需要用到所有的虚拟地址。
3. 进程一开始只需要建立最顶级的页表，只有当实际需要用到具体的虚拟地址时通过缺页中断（page fault）建立完整的映射。

缺页中断

在 Linux 中，用户进程的内存由 VMA 结构管理，虚拟地址到物理地址的转换过程中，会进行一系列检查：

1. 新申请的内存，由于 lazy 机制，只建立页表而没有真实物理内存的映射，此时页表里的权限是 R，访问时会发生缺页（page fault）中断。在缺页中断回调中，Linux 会去申请一页物理内存，并把页表权限设置为 R+W。这种不会产生 IO 操作的缺页中断，称为次缺页（minor page fault）。
2. 用户访问了非法的内存，MMU 也会触发缺页中断。在回调中检查发现当前进程并没有对应的虚拟内存地址的 VMA，给进程发送 SIGSEGV 信号报段错误并终止进程。
3. 代码段（Text）在 VMA 中权限为 R+X。如果程序中有野指针飞到此区域去写，则也会由 MMU 触发缺页中断，导致进程收到 SIGSEGV 信号。同理，如果 VMA 中权限为 R+W，而进程的 PC 指针飞到此区域去执行，同样会发生段错误。
4. 在代码段区域运行执行操作时发生缺页中断，说明该段代码数据未从硬盘加载。这种情况下，Linux 会申请一页物理内存，并从硬盘读取出代码段，此时产生了 IO 操作，为主缺页（ major page fault）。

缺页中断的过程大部分是由软件完成的，消耗时间比较久，是影响性能的一个关键指标。ps 命令可以查看某个进程的虚拟内存、物理内存的使用情况和缺页中断的次数。

```
ps -o vsz,rss,tsiz,dsiz,majflt,minflt,pmem -p <pid>
```

- vsz：虚拟内存的大小。
- rss：物理内存的大小。
- tsiz：程序代码占用的虚拟内存。
- dsiz：程序数据占用的虚拟内存。
- majflt：主缺页次数。
- minflt：次缺页次数。
- pmem：物理内存百分比。

TLB

多级页表虽然可以节省内存，但是也导致每次虚拟地址到物理地址的转换需要多次访问内存：64 位的四级页表，需要四次内存访问才能将虚拟地址转换成物理地址。

为了加快虚拟地址到物理地址的转换，利用程序的局部性，CPU 内部加入页表缓存——TLB（Translation Lookaside Buffer）。

perf 命令可以统计某个进程的 TLB 命中情况（dTLB 是数据内存页的 TLB；iTLB 是指令内存页的 TLB）：

```
$ perf stat -e dTLB-loads,dTLB-load-misses,iTLB-loads,iTLB-load-misses -p 15313
^C
 Performance counter stats for process id '15313':

     1,870,182,350      dTLB-loads
         5,471,552      dTLB-load-misses          #    0.29% of all dTLB cache
hits
         8,563,762      iTLB-loads
         3,321,761      iTLB-load-misses          #   38.79% of all iTLB cache
hits
```

Hugepages

Linux 默认的内存页大小一般都是 4KB。

```
$ getconf PAGESIZE
4096
```

而如今，几百 GB 内存的机器已是比较常见。4KB 的内存页，需要加载几千万个页表项，这会增加页表的内存开销和降低 TLB 的命中率。Linux 使用“大内存页”（hugepages）来优化大物理内存机器的虚拟内存管理。

大内存页和传统的 4KB 内存页是一种并列关系，而不是把 PAGESIZE 调大而已（直接修改Linux内核页面大小，涉及面较广，不一定合适。）。即，机器的内存一部分用大内存页进行管理，另一部分依然用 4KB 的大小进行管理。大内存页不可以被交换（swap）出内存。

通过启动大内存页，可以减少页表项的数量，从而减少维护它们的开销。同时增大 TLB 的覆盖范围，提高命中率。

可以通过在 /proc/meminfo 查看大内存页的情况：

```
$ grep Huge /proc/meminfo
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
```

```
FileHugePages:         0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:               0 kB
```

如上所示，默认的大内存页大小（Hugepagesize）为 2MB。大内存页还有另一种 1GB 大小的格式，适用于 TB 级别的内存。默认情况下，大内存页的数量（HugePages_Total）为 0。也就是说，没有开启大内存页。

编辑 /etc/sysctl.conf 文件，然后输入 `sysctl -p` 命令重新加载配置。

```
vm.nr_hugepages=126
```

设置 vm.nr_hugepages 之后，查看 /proc/meminfo：

```
$ grep Huge /proc/meminfo
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
FileHugePages:         0 kB
HugePages_Total:     126
HugePages_Free:      126
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:          258048 kB
```

Hugepages 的使用对上层应用不是透明的，需要在代码中指定，比如使用 mmap 的 MAP_HUGETLB 匿名映射 hugepages 内存页，或者通过 libhugetlbfs 使用 hugepages。

THP

前面说了，hugepages 的使用对上层应用不是透明的，需要修改应用代码。THP（Transparent Huge Pages，透明大页），顾名思义，就是对上层应用透明的大内存页。

查看本机的 THP 设置

```
$ cat /sys/kernel/mm/transparent_hugepage/enabled
always [madvise] never
```

- always：开启 THP。
- never：关闭 THP。
- madvise：仅在通过 `madvise()` 系统调用，并且设置了 `MADV_HUGEPAGE` 标记的内存区域中开启 THP。

查看本机/进程的 THP 占用

```
cat /proc/meminfo | grep AnonHugePages
cat /proc/$PID/smaps | grep AnonHugePages
```

分配 THP 的行为控制

```
$ cat /sys/kernel/mm/transparent_hugepage/defrag
always defer defer+madvise [madvise] never
```

- always：当系统分配不出透明大页时，触发内存回收和内存整理。内存回收和整理结束后，如果存在足够的连续空闲内存，则继续分配透明大页。
- defer：当系统分配不出透明大页时，转为分配普通的 4KB 页。同时唤醒 kswapd 内核进程进行后台内存回收，唤醒 kcompactd 内核进程进行后台内存整理。一段时间后，如果存在足够的连续空闲内存，khugepaged 内核守护进程将此前分配的 4KB 页合并为 2MB 的透明大页。
- madvise：仅在通过 madvise() 系统调用，并且设置了 MADV_HUGEPAGE 标记的内存区域中，内存分配行为等同于 always。其他情况为分配普通的 4KB 页。
- defer+madvise：仅在通过 madvise() 系统调用，并且设置了 MADV_HUGEPAGE 标记的内存区域中，内存分配行为等同于 always。其他情况的内存分配行为保持为 defer。
- never：禁止内存碎片整理。

khugepaged 内存碎片整理控制

```
$ cat /sys/kernel/mm/transparent_hugepage/khugepaged/defrag
1
```

- 0：关闭 khugepaged 内存碎片整理功能。
- 1：开启 khugepaged 内存碎片整理功能。khugepaged 内核守护进程会在系统空闲时周期性唤醒，尝试将连续的 4 KB 页合并成 2 MB 的透明大页。由于该操作会在内存路径中加锁，因此存在影响应用稳定性的可能性。

```
 $ cat /sys/kernel/mm/transparent_hugepage/khugepaged/alloc_sleep_millisecs
60000
```

- khugepaged 的重试间隔。当透明大页 THP 分配失败时，khugepaged 内核守护进程进行下一次大页分配前需要等待的时间，避免短时间内连续发生大页分配失败。默认值为 60000ms。

```
$ cat /sys/kernel/mm/transparent_hugepage/khugepaged/scan_sleep_millisecs
10000
```

- khugepaged 的唤醒间隔，默认值为 10000ms。

```
$ cat /sys/kernel/mm/transparent_hugepage/khugepaged/pages_to_scan
4096
```

- khugepaged 内核守护进程每次唤醒后扫描的页数。默认值为 4096 个页。

THP 的利弊

理论上，THP 能合并 4KB 的内存页，增加 TLB 命中的几率，使系统获得性能提升。

但是，THP 对内存碎片比较敏感，在内存紧张时，容易触发内存的直接回收或内存的直接整理，这两个操作都是同步等待的操作，会造成系统性能下降（抖动）。

另外，在 khugepaged 内核进程在进行内存合并操作时，会在内存路径中加锁，会对内存敏感型应用容易造成性能影响。

相比之下，hugepage 采用的是预留内存的方式，虽然使用上需要应用适配，但是性能和稳定性明显要好很多。

在 Redhat 的文档中，数据库类型的应用建议关闭 THP。

> However, THP is not recommended for database workloads.

Page Cache

free 命令

从 free 命令说起：

```
$ free -k
              total        used        free      shared  buff/cache   available
Mem:       16166056      897848      116196      590832    15152012    14563912
Swap:             0           0           0
```

free 是 Linux 下一个简单的常用命令，输出的信息如上。 Mem 一行数据描述了内存的使用情况。

- total：/proc/meminfo 中的 MemTotal，表示计算机的内存总量。
- used：被占用的内存，不能随时回收。
- free：完全没有被使用到的内存，/proc/meminfo 中的 MemFree。
- shared：/proc/meminfo 中的 Shmem，共享内存，一般是被 tmpfs 占用。tmpfs 的内存也是由 page cache 管理的，所以 buff/cache 的统计数据中也包含了这部分内存。tmpfs 占用的 page cache 不能随时回收。
- buff/cache：buffer + cache
    - buffer：/proc/meminfo 中的 Buffers。一般是块设备的读写缓存区。
    - cache：/proc/meminfo 中的 Cached，包括：
        - 文件系统的 cache。
        - slab 分配器管理的一些内核对象，比如页目录项缓存、inode 缓存。
- available：可用的内存（比较准确的评估值），其值为 /proc/meminfo 中的 MemAvailable。从上面的数据中，可以看出 available 的值略小于 free + buff/cache - shared。因此，buff/cache 对应的统计数据里面除了 shared 之外，还有一些其他不能被随时回收的内存。

drop caches

在 I/O 密集型的应用中，很容易遇到 buff/cache 占用量很高的情况（一般是 page cache 占用较多）。我们可以通过修改 /proc/sys/vm/drop_caches 的值来主动清理缓存（主动清理 page cache 的时候，如果存在大量脏页，可能引发大量 I/O）。

```
# 清除 page cache
echo 1 > /proc/sys/vm/drop_caches
# 回收 slab 分配器中的对象，比如目录项缓存、inode 缓存
echo 2 > /proc/sys/vm/drop_caches
# 清除 page cache 和 slab 分配器
echo 3 > /proc/sys/vm/drop_caches
```

刷新脏页

普通文件 I/O 模式下，write 系统调用在写入的数据到达 page cache 后就会返回成功，后续由内核线程异步将“脏页”刷新到硬盘（或者，程序可以调用 fsync 主动将数据同步到硬盘）。

内核控制脏页刷新到硬盘的相关参数有：

- /proc/sys/vm/dirty_background_ratio：当脏页占总内存的的百分比超过这个值时，后台线程开始刷新脏页。这个值如果设置得太小，可能不能很好地利用内存加速文件操作。如果设置得太大，则会周期性地出现一个写 I/O 的峰值。
- /proc/sys/vm/dirty_ratio：当脏页占用的内存百分比超过此值时，内核会阻塞掉写操作，并开始刷新脏页。
- /proc/sys/vm/dirty_background_bytes 和 /proc/sys/vm/dirty_bytes 是 dirty_background_ratio 和 dirty_ratio 按字节数绝对值限制的版本。_ratio 版本和 _bytes 版本只有一个能生效。如果设置了 _ratio 则 _bytes 自动变为 0，并失效，反之亦然。
- /proc/sys/vm/dirty_expire_centisecs：脏数据大约 dirty_expire_centisecs / 100 秒后会被刷新到硬盘。
- /proc/sys/vm/dirty_writeback_centisecs：dirty_writeback_centisecs / 100 秒唤醒一次刷新脏页的内核线程。

  说明： centy 中文意思是“百分之十”。

Page Cache 相关工具

- vmtouch 可以用来查看指定文件 page cache 使用情况，也可以手动将文件换入或换出缓存。

```
# 显示文件的 page cache 使用情况
$ vmtouch -v filename

# 换出文件的 page cache
$ vmtouch -ve filename

# 换入文件的page cache
$ vmtouch -vt filename
```

- nr_dirty 脏页数量和 nr_writeback 正在写回的脏页数量

```
$ cat /proc/vmstat | egrep "nr_dirty|nr_writeback"
nr_dirty 414
nr_writeback 0
```

内存回收和内存交换

Swap 分区

在内存充裕时，默认的 Linux 内核策略会比较激进地使用空闲内存缓存各种数据，以提高 I/O 性能。

而为了保证系统随时有足够的内存可以使用，Linux 内核需要在剩余内存较少时，对部分内存进行回收。一般情况下，可以直接回收缓存文件数据的 page cache —— 将脏页刷新到硬盘，然后回收内存即可。在内存比较紧张的时候，可能还需要把进程地址空间中的 heap、stack 等匿名页换出（swap）到硬盘上。

因为这些匿名内存页在硬盘上并没有对应的文件，Linux 内核通过 swap 机制在磁盘上开辟专用的 swap 分区来作为匿名页的 backing storage。Linux 中存在两种形式的 swap 分区：swap disk 和 swap file。前者是一个专用于做 swap 的块设备，作为裸设备提供给 swap 机制操作。后者则是存放在文件系统上的一个特定文件。

- `mkswap swapfile` 将一个 swapfile 转换为 swap 分区的格式。
- `swapon swapfile` 开启对应的 swap 分区。
- `swapon -s` 查看使用中的 swap 分区的状态。

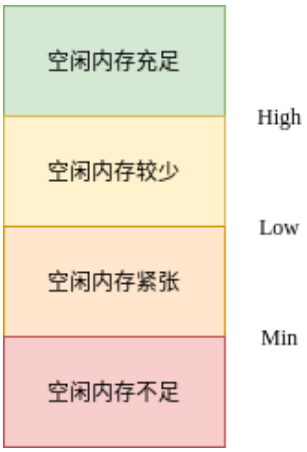
- `swapoff swapfile` 关闭对应的 swap 分区。
- `swapoff -a` 关闭 /proc/swaps 中的所有分区。

内存水位标记

从功能上讲，当内存压力较大（不够用）时，系统会将部分内存上的数据交换到 swap 空间上，以避免 OOM，而代价就是系统的 I/O 增加，处理速度会变慢。

那么 Linux 如何描述内存使用的压力呢？

Linux 使用内存水位标记的概念来描述内存使用的压力情况。Linux 为内存的使用设置了 3 个内存水位标记：high、low、min。



- 空闲内存在 high 以上表示空闲内存充足，内存使用的压力不大。
- 空闲内存在 high - low 的范围表示空闲内存较少，内存使用存在一定压力。
- 空闲内存在 low - min 的范围表示空闲内存紧张，内存使用压力较大。
- 空闲内存在 min 以下表示空闲内存严重不足。当空闲内存达到这个状态时，就说明内存面临很大压力。小于min这部分内存，内核是保留给特定情况下使用的，一般不会分配。
- 当空闲内存低于 low 的时候，内核的 kswapd 进程开始起作用，进行内存回收，以保证一般情况下，空闲内存尽可能够用。
- 当空闲内存小于等于 min 时，就会触发内存的直接回收，就是当内存分配时没有空闲内存可以满足要求时，触发直接内存回收（同步等待）。

内存回收和 swappiness

前面说了，内存回收有两条处罚路径：

1. 内核进程 kswapd 在后台进行内存回收。
2. 申请内存的时候，触发直接内存回收。

无论哪种内存回收方式，都需要解决一个问题：回收哪些内存？

从内核代码角度看，内存页主要有两种：匿名内存页和文件（缓存/映射）内存页。 文件内存页的回收方式是：脏页写回（writeback）+ 清空。 匿名内存页的回收方式是：swap。

内核通过 `/proc/sys/vm/swappiness` 参数来控制内存回收时如何在回收文件页和 swap 匿名页之间权衡。`swappiness` 的值越大，就会越积极使用 swap 匿名页的方式。默认值是60，可以的取值范围是0-100。

当 swappiness 取值为 0 时，不表示不会使用内存 swap。如果要禁止内存 swap，请使用 swapoff 命令关闭。

总结

本文主要介绍了一些基础的 Linux 内存相关的知识，包括：

1. 进程的地址空间。
2. 虚拟内存地址到内存物理地址的转换的基本原理。
3. Page cache 的基础知识。
4. 内存交换的基础知识。

当然还有很多方面没有涉及到，比如说 NUMA 相关的内存管理、malloc/free 的实现原理等等，以后有时间、有机会再写吧。