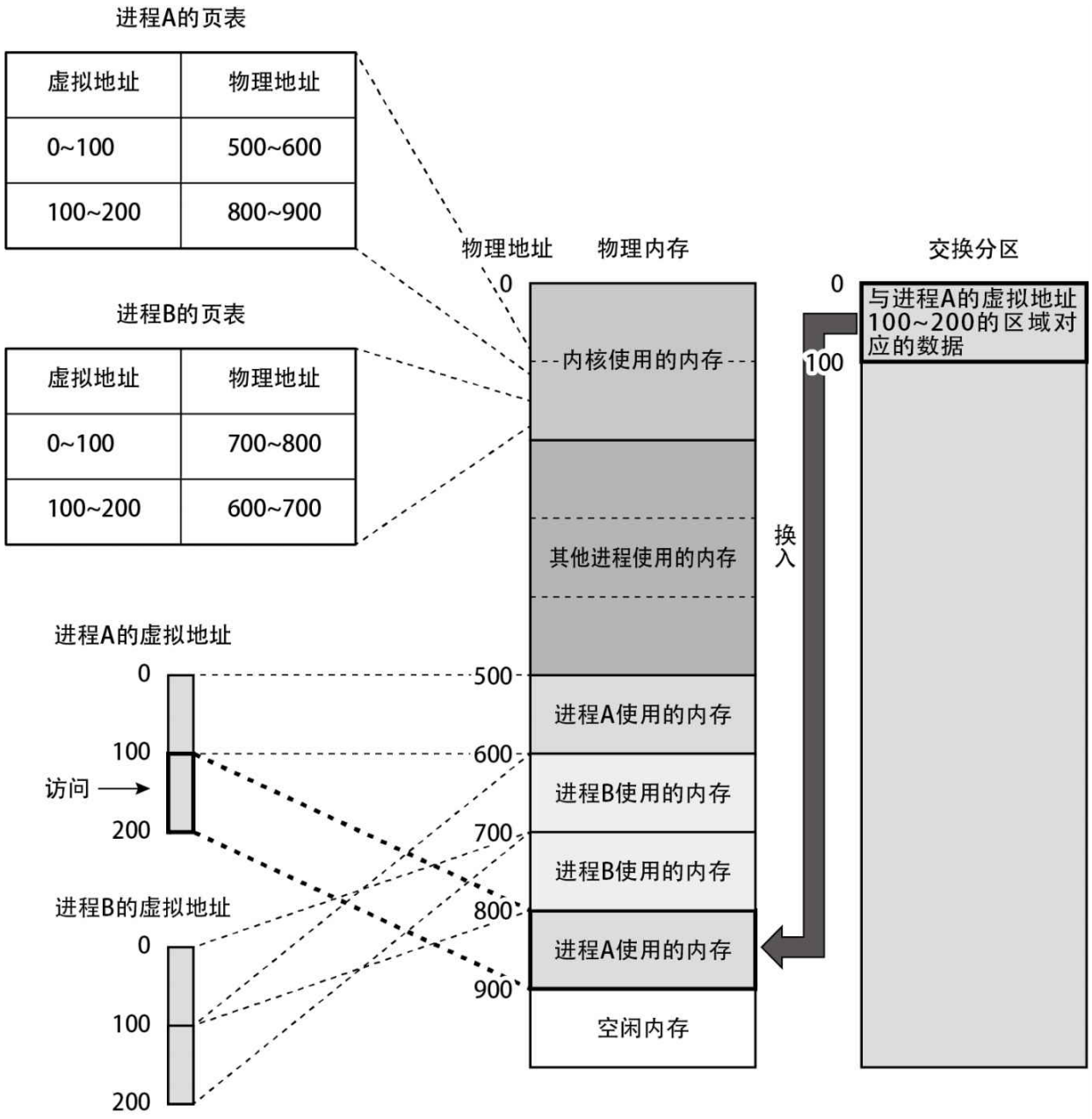

图 5-43　换入

换出与换入这两个处理统称为**交换**。在 Linux 中，由于交换是以页为单位进行的，所以也称为**分页**。同时，换入与换出也分别称为**页面调入**与**页面调出**。

Swap 乍看之下是一个能够使可用内存量扩充为“实际搭载的内存量 + 交换分区的容量”的美好机制，但这里其实存在一个非常大的陷阱。

那就是，相比对内存的访问速度，对普通的外部存储器的访问速度慢了几个数量级。

当系统长期处于内存不足时，访问内存的操作将导致页面不断地被换入和换出，从而导致系统陷入**系统抖动**（颠簸）状态。

大家或许经历过这样的情形：在使用笔记本时，明明没有进行读写文件的操作，但外部存储器的访问指示灯却亮着[5]。这种情形的原因大多在于系统抖动。一旦发生了抖动，系统就会暂时无法响应，然后一直保持那样的状态，最后宕机或者引发 OOM。

[5]如果外部存储器使用的是 HDD，计算机还会不停地发出“嘎吱嘎吱”的机器运行的声音。

如果系统频繁发生交换处理，就必须重新审视一下其设计是否存在问题。会引发系统抖动的系统不应当部署到服务器上。要调整设计，可以考虑降低系统负载以降低系统内存使用量，或者单纯地为系统增添内存等。

● 关于 Swap 的实验

可以通过 `swapon --show` 命令查看系统交换分区的信息。

```
$ swapon --show
NAME           TYPE            SIZE     USED     PRIO
devsdd7     partition        954M       0B       -1
$
```

在笔者的计算机上，*dev*sdd7 分区被用作交换分区，大小约为 1 GB。交换分区的大小可以通过 `free` 命令查看。

```
$ free
        total    used     free  shared buff/cache  available
Mem: 32942008 1864144 21640824  298120    9437040   30135580
Swap:  976892       0   976892
$
```

在输出内容中，Swap: 这一行即为交换分区的信息。total 字段下的 976892 为交换分区的大小，单位为 KB。在笔者的计算机中，共有约 32 GB 的物理内存和约 1 GB 的交换分区，所以共有约 33 GB

的可用内存。used 字段的值为 0，这代表执行 free 命令时完全没用上交换分区。

在系统运行期间，定期通过 sar -W 命令确认系统中是否发生了交换处理是一个不错的习惯。

```
$ sar -W 1
( 略 )
23:30:00     pswpin/s pswpout/s
23:30:01         0.00      0.00
23:30:02         0.00      0.00
23:30:03         0.00      0.00
( 略 )
```

其中，pswpin/s 字段为每秒发生换入的次数，pswpout/s 字段为每秒发生换出的次数。如果系统性能突然下降，并且这两个字段出现非零的数值，那就要考虑发生系统抖动的可能性了。

在此基础上执行 sar -S 命令，则当发生交换处理时，就可以确认该交换处理到底是暂时性的（马上会结束）还是毁灭性的（引发系统抖动）。

```
$ sar -S
( 略 )
23:28:59   kbswpfree  kbswpused  %swpused  kbswpcad  %swpcad
23:29:00      976892          0      0.00         0     0.00
23:29:01      976892          0      0.00         0     0.00
23:29:02      976892          0      0.00         0     0.00
23:29:03      976892          0      0.00         0     0.00
( 略 )
```

基本上，只需要通过 kbswpused 字段的值了解交换分区使用量的变化趋势即可。如果这个值不断增加，就非常危险。

这里补充说明一下前面提起过的硬性页缺失与软性页缺失。

类似于交换这类需要访问外部存储器的缺页中断称为**硬性页缺失**。与此相对，无须访问外部存储器的缺页中断称为**软性页缺失**。虽然无论发生硬性页缺失还是软性页缺失，都需要内核进行处理，进而影响性能，但硬性页缺失所产生的影响要更大。

5.16 多级页表

我们在前面提到过，在进程的页表中保存着表示虚拟地址空间中的页面是否关联着物理内存的数据。下面一起来思考一下如何以最简单的单层结构实现这个页表。

在 x86_64 架构上，虚拟地址空间大小为 128 TB[6]，页面大小为 4 KB，页表项的大小为 8 字节。通过上面的信息可以算出，一个进程的页表就需要占用 256 GB 的内存（= 8 B×128 TB / 4 KB）。以笔者的计算机为例，由于只有 32 GB 的内存，所以一个进程也无法创建。

> [6]最近出现了拥有更大虚拟地址空间的 CPU，并且 Linux 也对其进行了适配。但为了便于说明，本书不考虑这部分 CPU。

为了避免这样的情况，x86_64 架构的页表采用多级结构，而非上面描述的单层结构。这样就能节约大量的内存。

在现实中，x86_64 架构的页表结构非常复杂，因此在比较单层页表与多级页表的不同时，我们将利用比实际结构简单的模型。假设一个页面大小为 100 字节，虚拟地址空间的大小为 1600 字节。

在这种情况下，如果进程仅使用 400 字节的物理内存，那么单层页表如图 5-44 所示。

虚拟地址	物理地址
0~100	300~400
100~200	400~500
200~300	500~600
300~400	600~700
400~500	×
500~600	×
600~700	×
700~800	×
800~900	×
900~1000	×
1000~1100	×
1100~1200	×
1200~1300	×
1300~1400	×
1400~1500	×
1500~1600	×

图 5-44　单层页表

假设多级页表为每 4 个页面归为 1 组的 2 级结构，则该页表如图 5-45 所示。

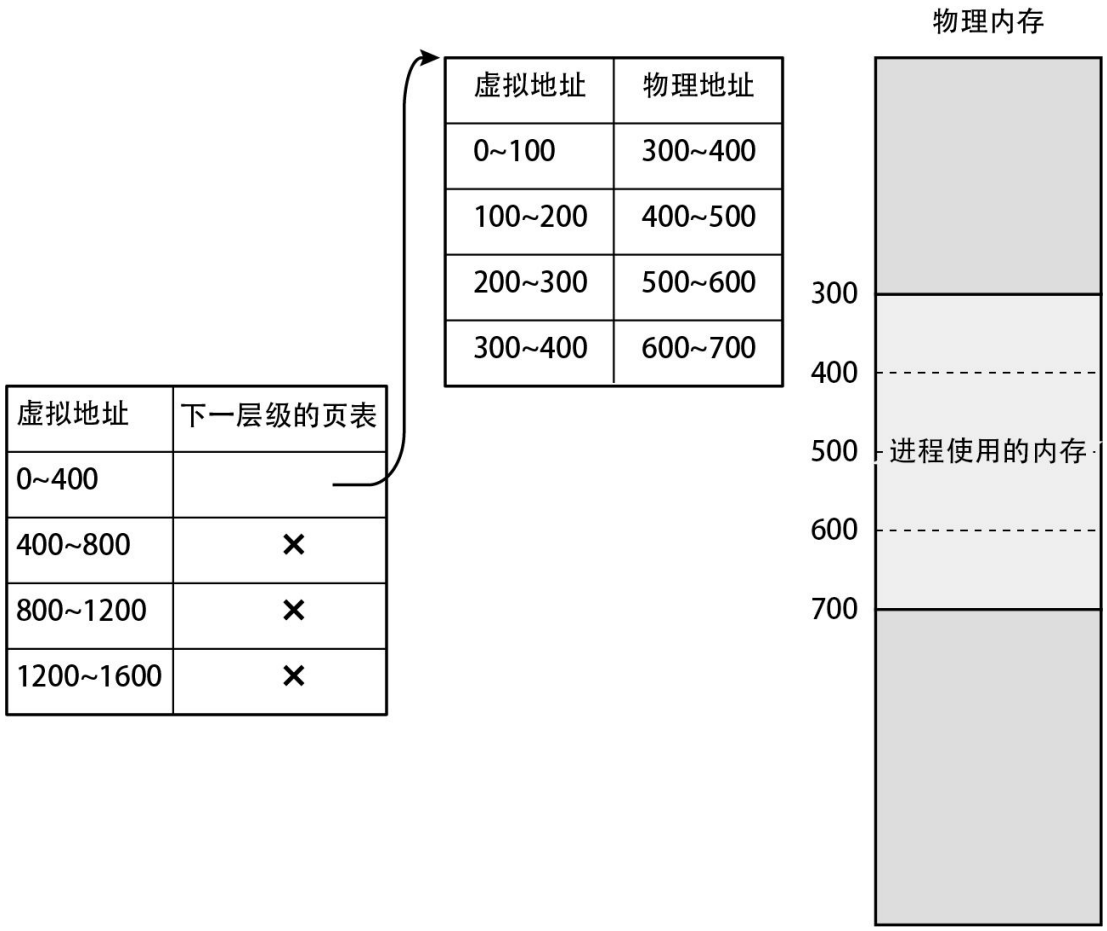

图 5-45　多级页表

可以看到，页表项的数量由 16 个减少到 8 个。但随着虚拟内存使用量增加，页表的使用量也会增加，如图 5-46 所示。

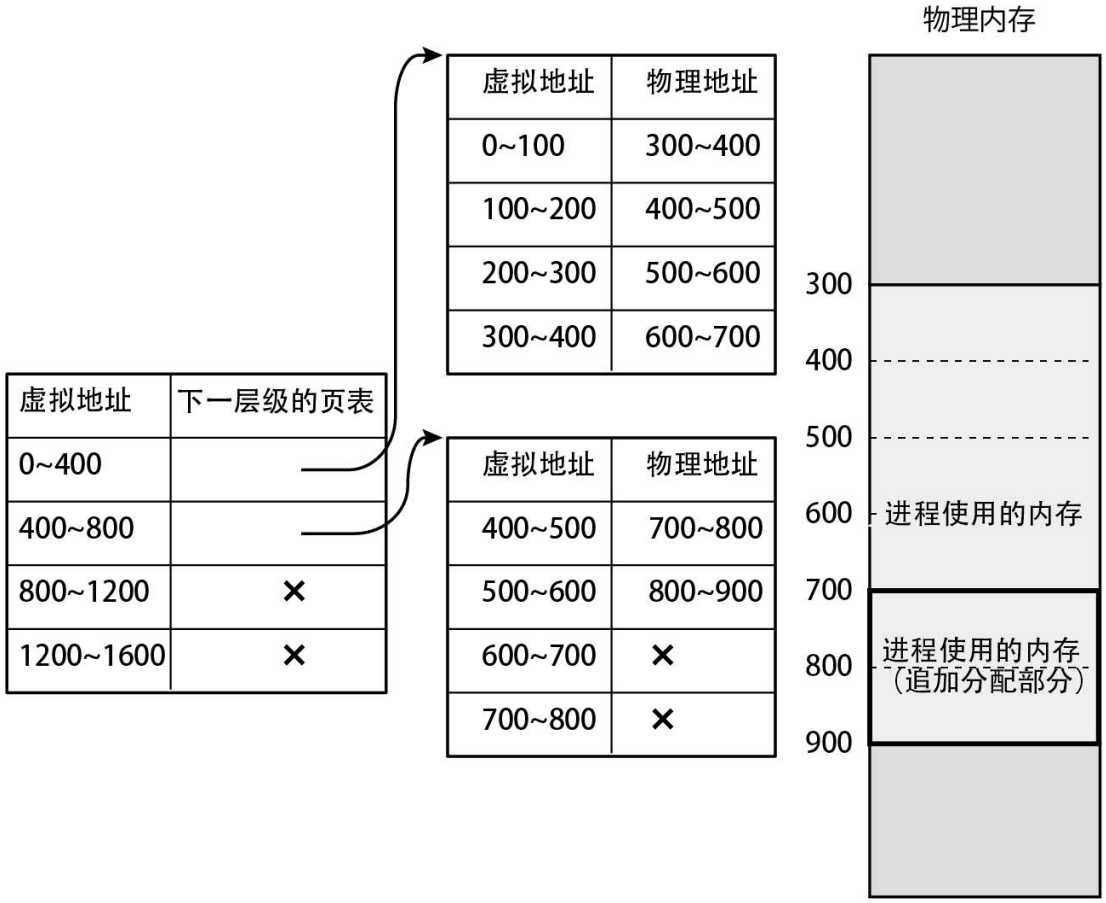

图 5-46　随着虚拟内存使用量增加，页表的内存量也随之增加

当虚拟内存使用量增加到一定程度时，多级页表的内存使用量就会超过单层页表。但这种情况非常罕见，所以从系统整体来看，也是多级页表的内存使用量更少。

另外，x86_64 架构的页表结构还要更加复杂，达到了 4 级。但是，这部分内容超出了本书的范围，因此这里不再介绍。另外，为了作图方便，在不涉及多级页表相关的内容时，本书将像前几章一样，画成单层页表的形式。

我们可以通过 `sar -r ALL` 命令中的 `kbpgtbl` 字段查看页表所使用的物理内存量。

```
$ sar -r ALL 1
( 略)
23:40:05 kbmemfree kbmemused  %memused  kbbuffers  kbcached ↵
kbcommit   %commit  kbactive   kbinact    kbdirty  kbanonpg ↵
```

```
kbslab    kbkstack   kbpgtbl  kbvmused
23:40:06  21614072  11327936     34.39        6084   8897948 ↲
8104164      23.89   8943372   1556624         248   1548816 ↲
560512       14352     53944         0
23:40:07  21613884  11328124     34.39        6084   8897944 ↲
8104164      23.89   8944004   1556620         248   1549572 ↲
560512       14320     54044         0
23:40:08  21613332  11328676     34.39        6084   8897944 ↲
8104164      23.89   8944464   1556620         248   1550056 ↲
560512       14336     54184         0
( 略)
```

除了分配过多物理内存给进程之外，“创建太多进程”以及“进程使用太多虚拟内存而导致页表占用的区域增加”等情况都会使系统陷入内存不足。前者可以通过降低进程的并发量，或者减少系统上同时运行的进程数量等方式来缓解，后者则可以通过下面介绍的标准大页来应对。

5.17 标准大页

如上一节所述，随着进程的虚拟内存使用量增加，进程页表使用的物理内存量也会增加。

此时，除了内存使用量增加的问题之外，还存在 fork() 系统调用的执行速度变慢的问题，这是因为 fork() 是通过写时复制创建进程的，这虽然不需要复制物理内存的数据，但是需要为子进程复制一份与父进程同样大小的页表。为了解决这个问题，Linux 提供了**标准大页**机制。

顾名思义，标准大页是比普通的页面更大的页。利用这种页面，能有效减少进程页表所需的内存量。

下面，我们以每页 100 字节，每级 400 字节的 2 级结构的页表（图 5-45）为例来进行说明。该 2 级页表的所有页面都被分配物理内存时的情形如图 5-47 所示。

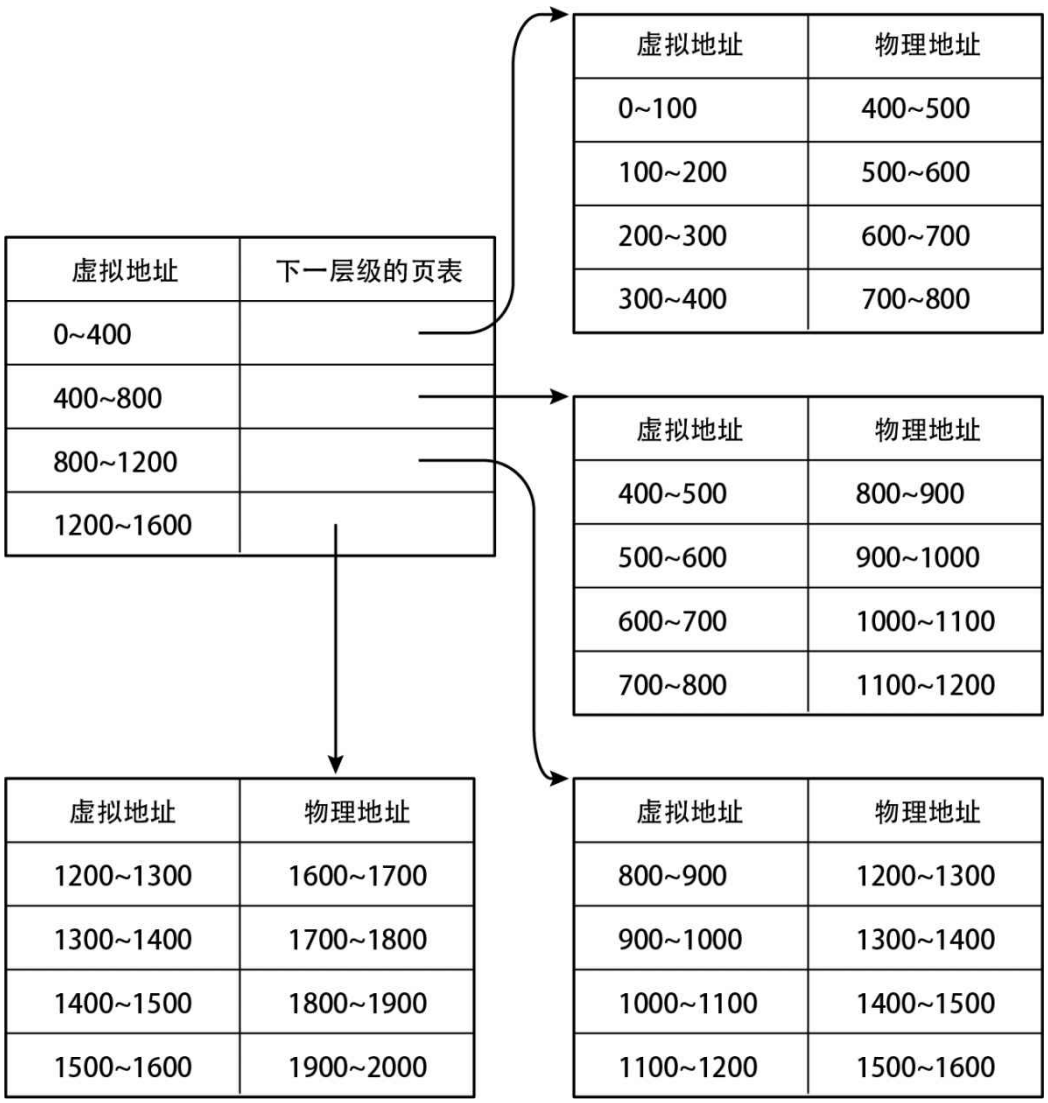

图 5-47　所有页面都被分配物理内存时的情形

把其中的页面置换成大小为 400 字节的标准大页后，页表将减少一个层级，如图 5-48 所示。

虚拟地址	物理地址
0~400	400~800
400~800	800~1200
800~1200	1200~1600
1200~1600	1600~2000

图 5-48　用标准大页置换后的页表

可以看到，页表项的数量由 20 个减少到 4 个。像这样，通过将普通页面置换成标准大页，不但能降低页表的内存使用量，还能提高 fork() 系统调用的执行速度。

虽然 x86_64 架构中的标准大页比图 5-48 的结构要复杂得多，但现在并不需要在意这一点。只需要记住标准大页可以令使用大量虚拟内存的进程减少页表的内存开支即可。

● 标准大页的用法

在 C 语言中，通过为 mmap() 函数的 flags 参数赋予 MAP_HUGETLB 标志，可以获取标准大页。但在实际应用中，比起让编写的程序直接获取标准大页，更常用的做法是为现有程序开启允许使用标准大页的设置。

数据库与虚拟机管理器等都是需要使用大量虚拟内存的软件，它们会提供关于标准大页的设置项，请根据实际情况决定使用与否。通过进行设置，能够减少这类软件的内存使用量，同时还能提高 fork() 的执行速度。

● 透明大页

Linux 上还存在一个名为**透明大页**的机制。当虚拟地址空间内连续多个 4 KB 的页面符合特定条件时，通过透明大页机制能将它们自动转换成一个大页。

乍看之下这是非常便利的功能，但也存在一些问题，例如将多个页面汇聚成一个大页的处理，以及当不再满足上述条件时将大页重新拆分为多个 4 KB 的页面的处理等，会引起局部性能下降。为此，在搭建系统时，我们有时会禁用透明大页。

通过读取 *sys*kernel/mm/transparent_hugepage/enabled 文件的内容，即可获知系统是否启用了透明大页。在 Ubuntu 16.04 上默认是启用的（always 表示启用）。

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

如果希望禁用透明大页，只需往该文件写入 never 即可。

```
$ sudo su
# echo never >sys/kernel/mm/transparent_hugepage/enabled
#
```

顺便一提，当设定为 madvise 时，表示仅对由 madvise() 系统调用设定的内存区域启用透明大页机制。

第 6 章　存储层次

大家见过图 6-1 这种展示计算机的存储器的层次结构的图吗？

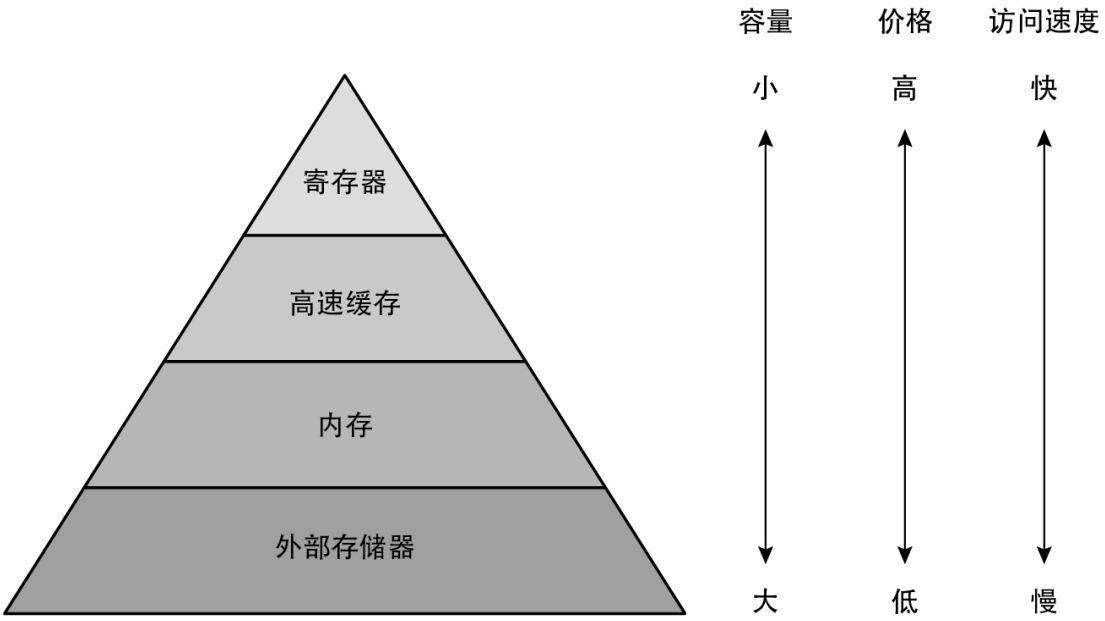

图 6-1　存储器的层次结构

图 6-1 中展示了计算机上的各种存储器。一方面，越靠近顶层，设备的容量越小，单位容量的价格越高；另一方面，越靠近顶层，访问速度就越快。本章将针对这些存储器，讨论以下两点内容。

- 在容量和性能方面，各种存储器存在多大的差距呢？[1]
- 硬件和 Linux 是如何在考虑了这些差距的前提下运行的呢？

[1]由于部分存储器的具体价格难以计算，所以本书将不讨论价格方面的内容。

6.1 高速缓存

这里再介绍一次计算机的运作流程（省略了从内存读取指令的部分）。

① 根据指令，将数据从内存读取到寄存器。

② 基于寄存器上的数据进行运算。

③ 把运算结果写入内存。

就近期的硬件而言，与在寄存器上执行运算所耗费的平均时间相比，访问内存会消耗更多的时间，产生更长的延迟。以笔者的计算机为例，前者执行一次的时间还不到 1 纳秒，但后者执行一次的时间能达到几十纳秒[2]。对于计算机系统来说，无论流程②的处理速度有多快，流程①和流程③都会成为性能瓶颈，因此整体的处理速度将受限于内存的访问延迟。

> [2]在不同的硬件上，这两者的速度会存在巨大的差距，因此请重点关注它们的相对差距。

而高速缓存的存在，正是为了抹平寄存器与内存之间的性能差距。

从高速缓存到寄存器的访问速度比从内存到寄存器的访问速度快了几倍甚至几十倍，利用这一点，即可提高流程①和流程③的处理速度。高速缓存通常内置于 CPU 内，但也存在位于 CPU 外的类型。

接下来，让我们看一下高速缓存的运作方式。首先需要注意的是，此后叙述的与高速缓存相关的内容皆止步于硬件设备层面，不会涉及内核[3]。

> [3]准确来说，因为需要在预定时间点销毁高速缓存等，所以大部分情况下高速缓存由内核控制，但本书不讨论这部分内容。

在从内存往寄存器读取数据时，数据先被送往高速缓存，再被送往寄存器。所读取的数据的大小取决于**缓存块大小**（cache line size）的值，该值由各个 CPU 规定。

假设缓存块的大小为 10 字节，高速缓存的容量为 50 字节，并且存在两个长度为 10 字节的寄存器（R0 与 R1）。在这样的运行环境下，把内存地址 300 上的数据读取到 R0 时的情形如图 6-2 所示。

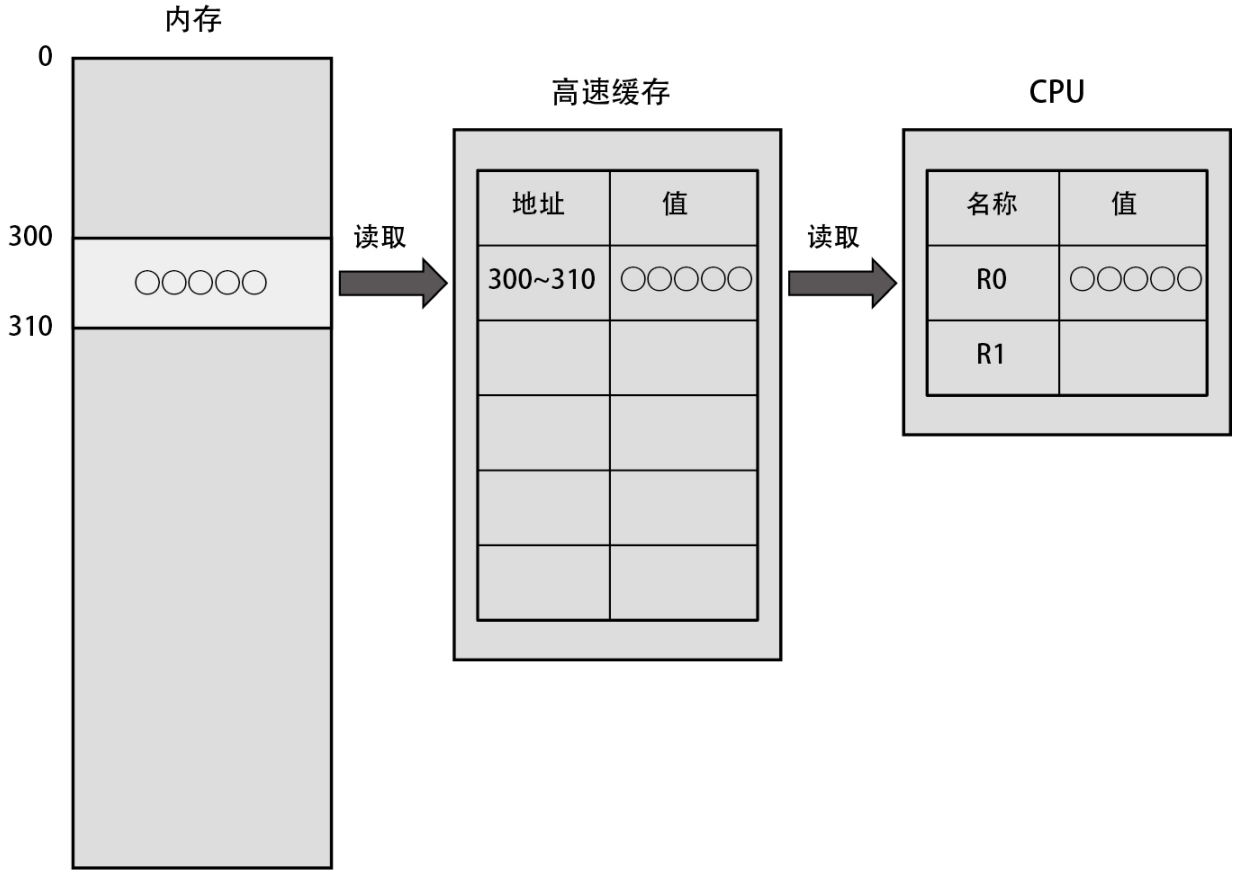

图 6-2　将内存地址 300 上的数据读取到 R0

此后，当 CPU 需要再次读取地址 300 上的数据时，比如需要再次把同样的数据读取到 R1 时，将不用从内存读取数据，只需读取已经存在于高速缓存上的数据即可，如图 6-3 所示。