```
0        2        2
0        3        3
0        4        4
0        5        5
( 略 )
1        195      96
1        196      97
1        197      98
1        198      99
1        199      100
$
```

图 4-37 所示为 2 个进程在逻辑 CPU0 上的运行情况。

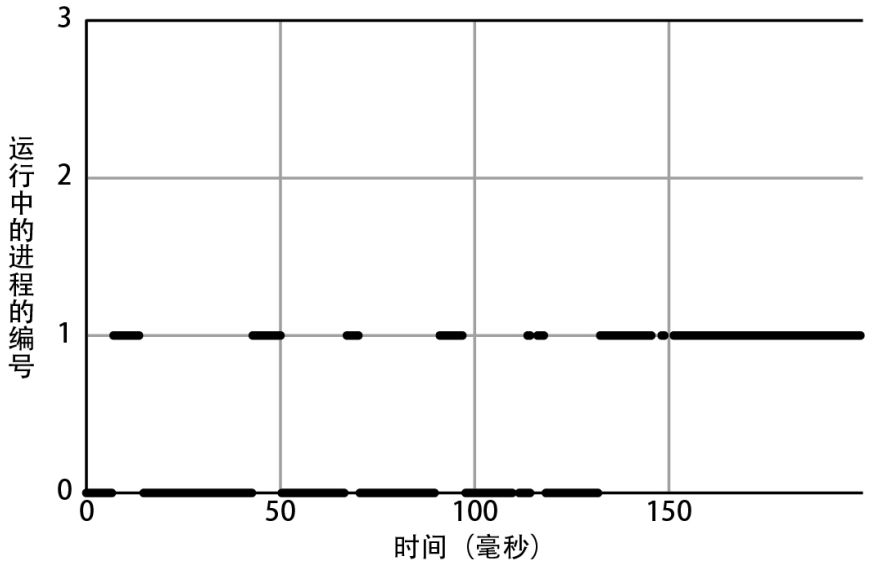

图 4-37　在逻辑 CPU0 上运行的进程

可以看到，优先级高（nice 的值更小）的进程 0，比优先级低（nice 的值更大）的进程 1 获得了更多的 CPU 时间。

2 个进程的进度如图 4-38 所示。

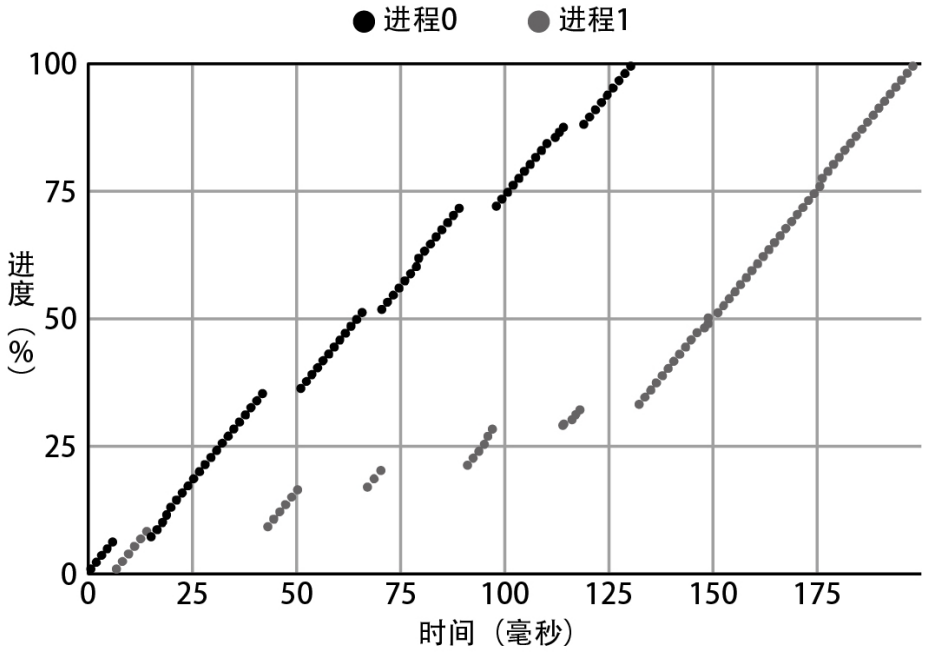

图 4-38　进程 0 与进程 1 的进度

可以看到，由于进程 0 优先被分配 CPU 时间，所以它比进程 1 更早结束运行，而进程 1 在进程 0 结束后继续运行到结束。由于处理量并没有改变，所以 2 个进程的处理依旧需要消耗 200 毫秒，这和没有设置优先级时是一样的。

优先级除了可以通过在程序中调用函数来设定，还能通过执行 nice 命令直接设定。通过 nice 命令的 -n 选项指定优先级，便可令某个程序以指定的优先级运行。通过这种方式，就能在不修改源代码的情况下方便地更改程序的优先级。

下面展示了如何令 echo hello 命令在优先级为 5 的状态下运行。

```
$ nice -n 5 echo hello
Hello
$
```

顺便一提，在 sar 命令的输出中，%nice 字段的值表示在从默认值 0 更改为其他优先级后，进程运行时间所占的比例。下面，我们来看一下 loop.py 程序在降低优先级的状态下运行时 sar 的输出。

```
$ nice -n 5 python3 ./loop.py &
[1] 17831
$ sar -P ALL 1 1
( 略)
18:28:27  CPU  %user   %nice %system %iowait %steal   %idle
18:28:28  all   0.25   12.52    0.00    0.00   0.00   87.23
18:28:28    0   0.00  100.00    0.00    0.00   0.00    0.00
18:28:28    1   1.00    0.00    0.00    0.00   0.00   99.00
18:28:28    2   0.00    0.00    0.00    0.00   0.00  100.00
18:28:28    3   0.00    0.00    0.00    0.00   0.00  100.00
18:28:28    4   0.00    0.00    0.00    0.00   0.00  100.00
18:28:28    5   0.99    0.00    0.00    0.00   0.00   99.01
18:28:28    6   0.00    0.00    0.00    0.00   0.00  100.00
18:28:28    7   0.00    0.00    0.00    0.00   0.00  100.00
( 略 )
$
```

在不使用 nice 命令时，%user 的值为 100；与此相对，在使用 nice 命令后，%nice 的值变成了 100。

在测试结束后，记得结束正在运行的进程。

```
$ kill 17831
```

除此之外，本章中使用的 taskset 命令也是 OS 提供的调度器相关的程序。该命令请求被称为 sched_setaffinity() 的系统调用，以将程序限定在指定的逻辑 CPU 上运行。

第 5 章　内存管理

Linux 通过内核中名为**内存管理系统**的功能来管理系统上搭载的所有内存（图 5-1）。除了各种进程以外，内核本身也需要使用内存。

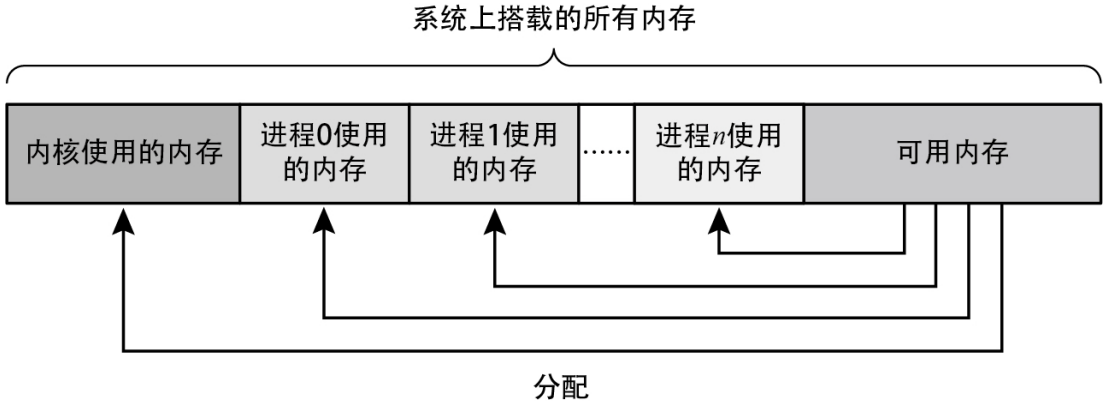

图 5-1　内存管理系统管理着所有内存

5.1 内存相关的统计信息

可以通过 free 命令获取系统搭载的内存总量和已消耗的内存量。

```
$ free
              total        used        free      shared  buff/cache   available
Mem:       32942000      337640    30641272       18392     1963088    32000464
Swap:             0           0           0
$
```

这里对 Mem: 这一行中的重要字段进行说明。需要注意的是，上面的所有数值的单位都为千字节（KB）。

- **total** 字段：系统搭载的物理内存总量。在上面的例子中约为 32 GB
- **free** 字段：表面上的可用内存量（详情请参考下面的 **available** 字段的说明）
- **buff/cache** 字段：缓冲区缓存与页面缓存（详见第 6 章）占用的内存。当系统的可用内存量（**free** 字段的值）减少时，可通过内核将它们释放出来
- **available** 字段：实际的可用内存量。本字段的值为 **free** 字段的值加上当内存不足时内核中可释放的内存量。“可释放的内存”指缓冲区缓存与页面缓存中的大部分内存，以及内核中除此以外的用于其他地方的部分内存

关于 Swap: 这一行的内容，后面将会具体说明。

接下来，我们通过图 5-2 来看一下 free 命令中的各个字段。

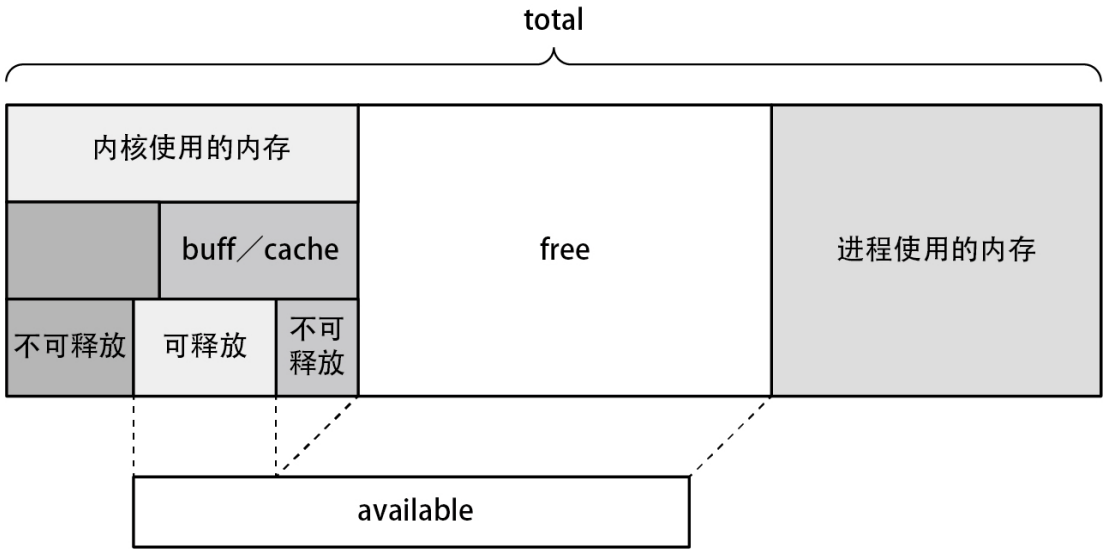

图 5-2　可以通过 free 命令确认的信息

另外，使用 sar -r 命令，即可通过其第 2 个参数指定采集周期（在下面的例子中为 1 秒），对内存的相关信息进行采集。

```
$ sar -r 1
( 略 )
08:19:40 kbmemfree   kbmemused  %memused kbbuffers  kbcached ↲
kbcommit   %commit   kbactive   kbinact   kbdirty
08:19:41  28892368    4049632     12.29      5980   3117188 ↲
 2127556      6.46    2413616    937524       112
08:19:42  28892368    4049632     12.29      5980   3117188 ↲
 2127556      6.46    2413616    937524       112
08:19:43  28892368    4049632     12.29      5980   3117188 ↲
 2127556      6.46    2413616    937524       112
08:19:44  28892368    4049632     12.29      5980   3117188 ↲
 2127556      6.46    2413616    937524       112
( 略 )
```

下表列出了 free 命令与 sar -r 命令中相对应的字段。

free 命令的字段	**sar -r** 命令的字段
total	不存在

free 命令的字段	**sar -r** 命令的字段
free	kbmemfree
buff/cache	kbbuffers + kbcached
available	不存在

5.2　内存不足

如图 5-3 所示，随着内存使用量增加，可用内存变得越来越少。

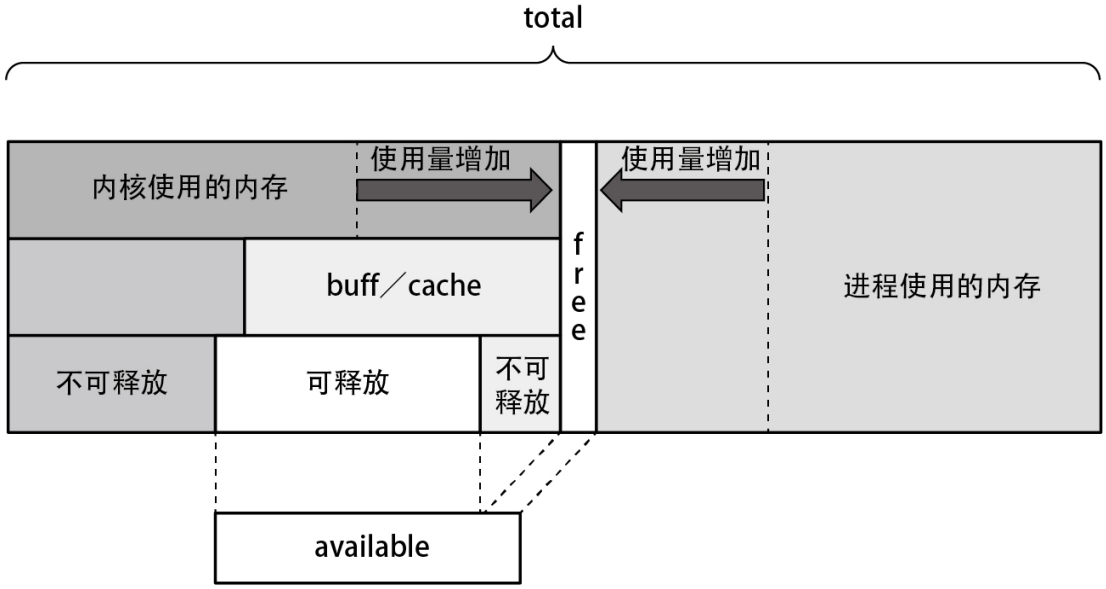

图 5-3　随着内存使用量增加，可用内存越来越少

在变成图 5-3 这样的状态后，内存管理系统将回收内核中可释放的内存[1]（图 5-4）。

[1]为了便于说明，这里解释为一次性回收所有可释放的内存，但在现实中，回收逻辑要更加复杂一些。

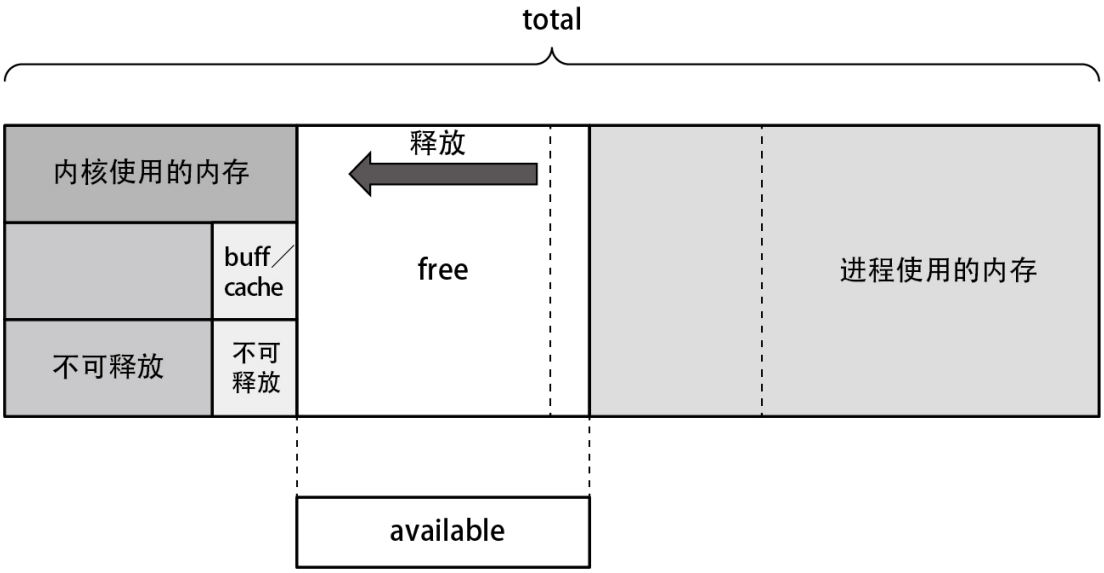

图 5-4　释放内核中的内存区域

如果内存使用量继续增加，系统就会陷入做什么都缺乏足够的内存，以至于无法运行的**内存不足**（Out Of Memory，OOM）状态（图 5-5）。

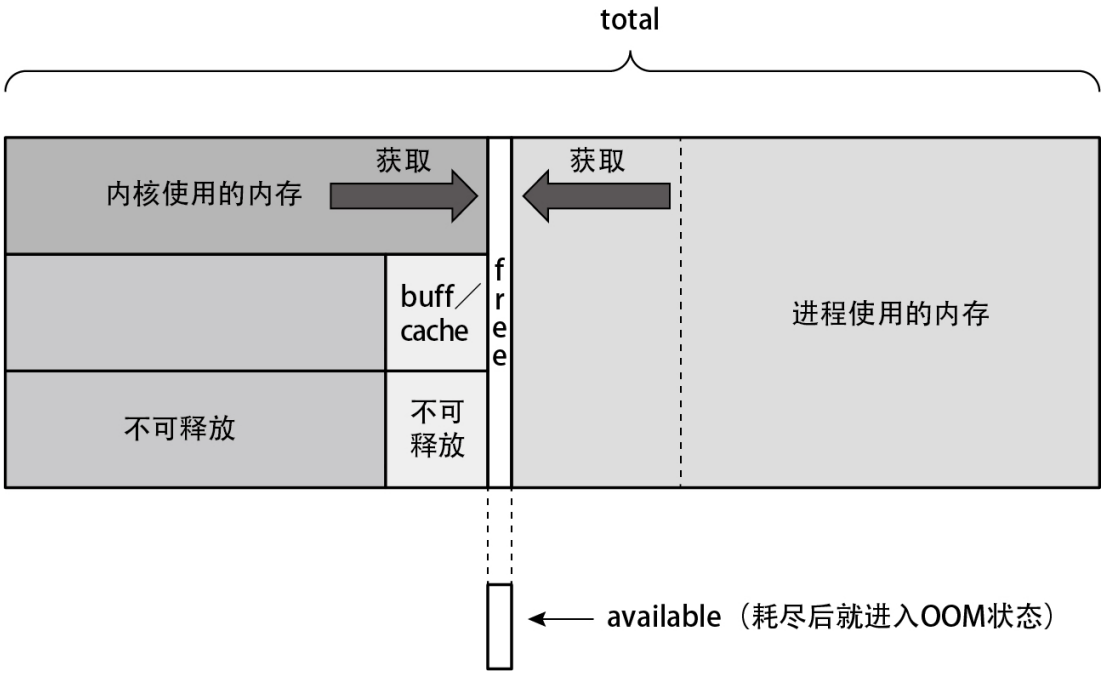

图 5-5　OOM 状态

在进入 OOM 状态后，内存管理系统会运行被称为 OOM killer 的可怕功能，该功能会选出合适的进程并将其强制终止（kill 掉），以释放出更多内存（图 5-6）。

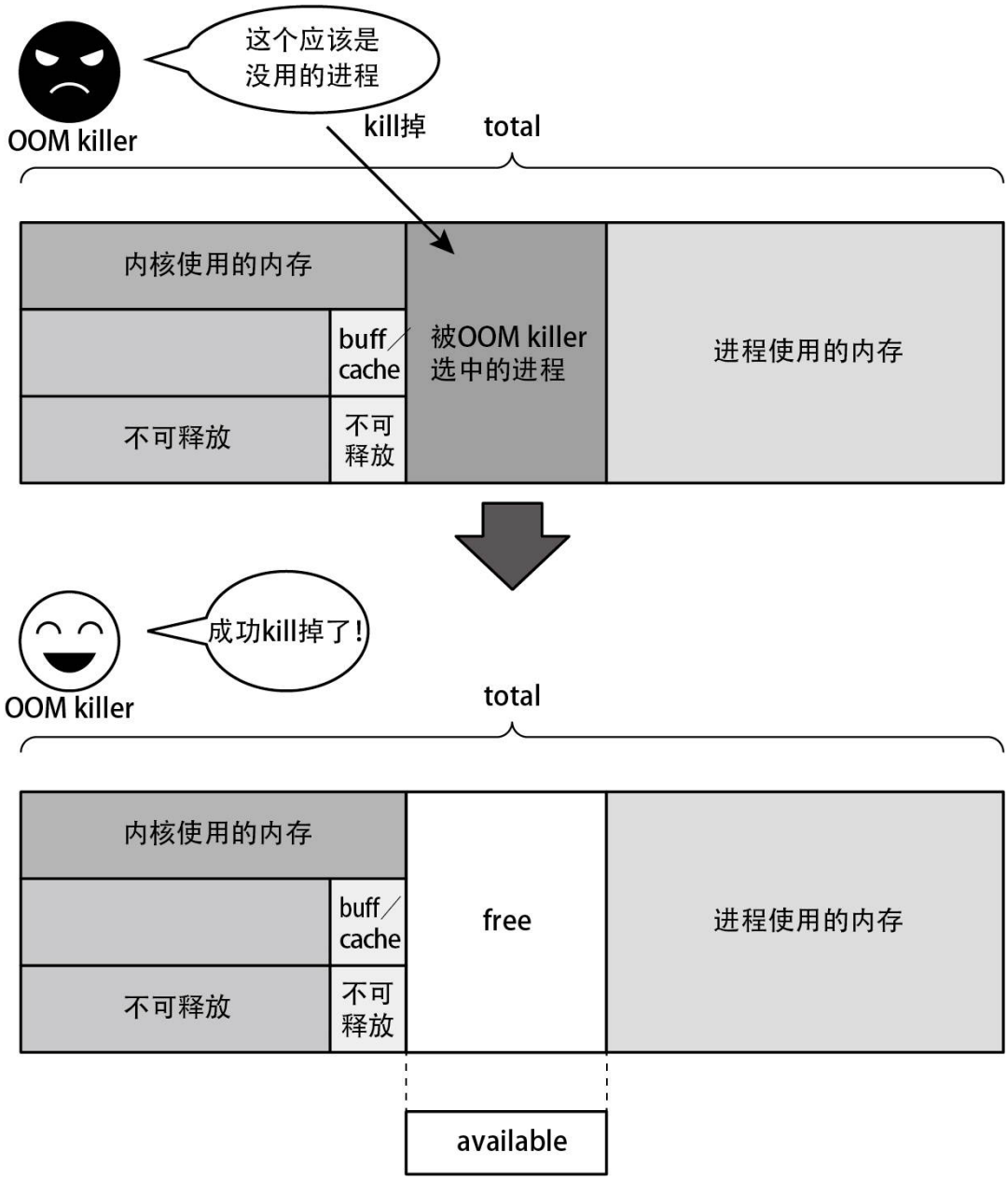

图 5-6　OOM killer 将被选中的进程 kill 掉

如果是个人计算机，这可能并非什么大问题；但如果是商用服务器，则完全不知道结束的是哪一个进程，这种状态非常令人困扰。虽然有办法令特定进程排除在 OOM killer 的选择范围之外，但是要在业务

用的进程中筛选出允许强制结束的进程是非常困难的。因此，也有将服务器上的 sysctl 的 vm.panic_on_oom 参数从默认的 0（在发生 OOM 时运行 OOM killer）变更为 1（在发生 OOM 时强制关闭系统）这样的做法。

5.3 简单的内存分配

现在开始解释内存管理系统的**内存分配机制**。要想理解现实中 Linux 的内存分配机制，就必须理解虚拟内存。本章后面会对虚拟内存进行介绍，现在我们先抛开虚拟内存的知识，只说明简单的分配机制以及其中存在的问题（这些问题都能通过虚拟内存解决）。

内核为进程分配内存的时机大体上分为以下两种。

- **在创建进程时**
- **在创建完进程后，动态分配内存时**

其中，创建进程时的内存分配已经在第 3 章解释过了，所以这里直接从创建完进程后的动态内存分配开始说明。

在进程被创建后，如果还需要申请更多内存，那么进程将向内核发出用于获取内存的系统调用，提出分配内存的请求。内核在收到分配内存的请求时，会按照请求量在可用内存中分出相应大小的内存，并把这部分内存的起始地址返回给提出请求的进程。图 5-7 所示为进程请求 100 字节（Byte，简称 B）的内存时的情形。

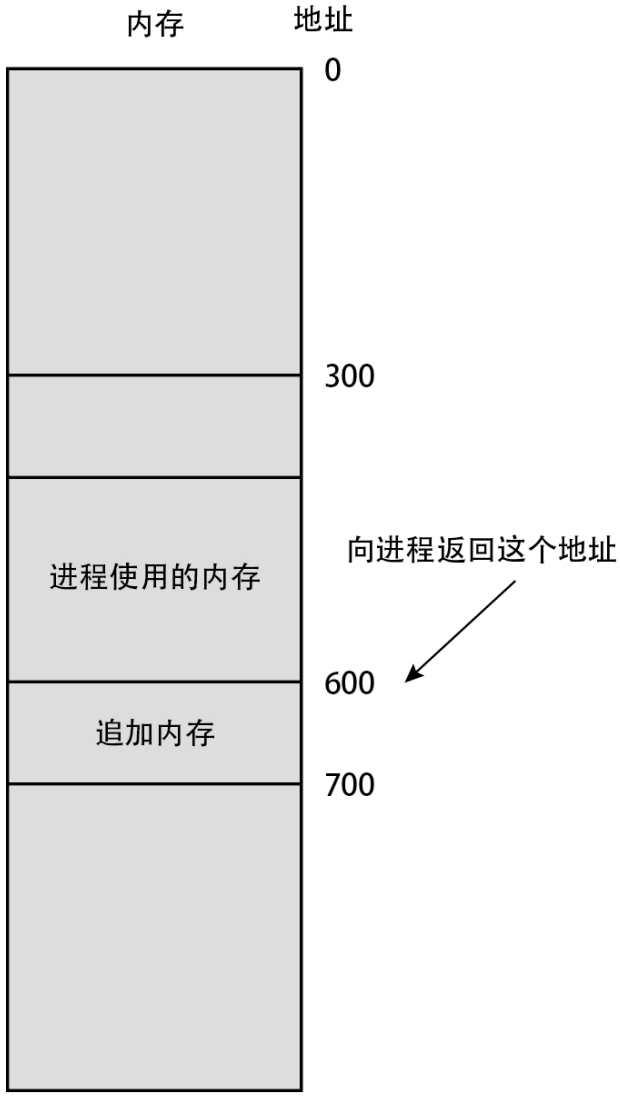

图 5-7　进程请求 100 字节的内存

但是，这种分配方式会引起下列问题。

- 内存碎片化
- 访问用于其他用途的内存区域
- 难以执行多任务

接下来，我们依次进行详细说明。

● 内存碎片化

在进程被创建后，如果不断重复执行获取与释放内存的操作，就会引发内存碎片化的问题。如图 5-8 所示，300 字节的可用内存分散在 3 个不同的位置，大小分别为 100 字节，这就导致无法分配 100 字节以上的内存区域。

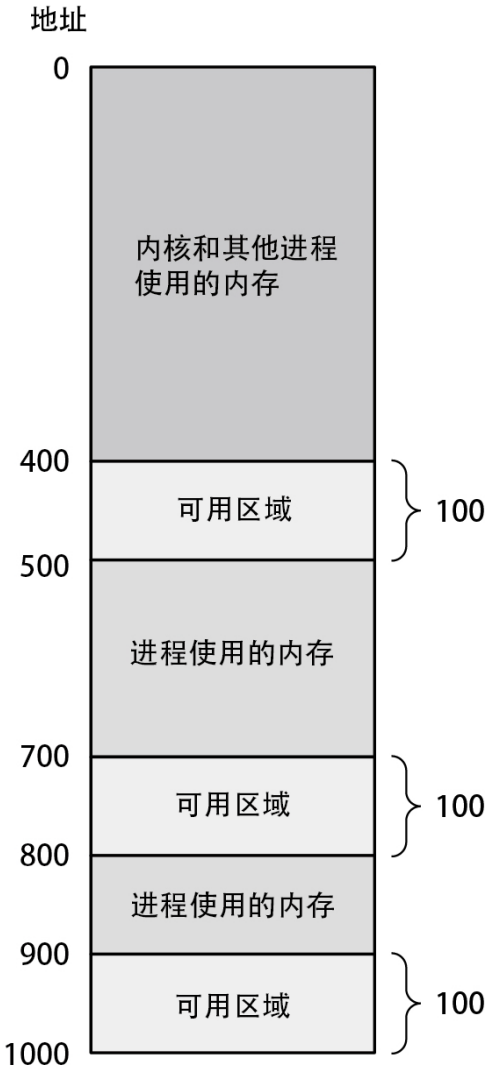

图 5-8　内存碎片化

大家或许会想：只要把这 3 个内存区域当作 1 组来使用就没问题了吧？但这是无法实现的，原因如下。

- 进程在每次获取内存后，都需要知道获取的这部分内存涵盖多少个区域，否则就无法使用这些内存，这很不方便

- 进程无法创建比 100 字节更大的数据块，例如 300 字节的数组等

● 访问用于其他用途的内存区域

到目前为止，在我们介绍的简单的机制中，进程均可通过内存地址来访问内核或其他进程所使用的内存（图 5-9）。

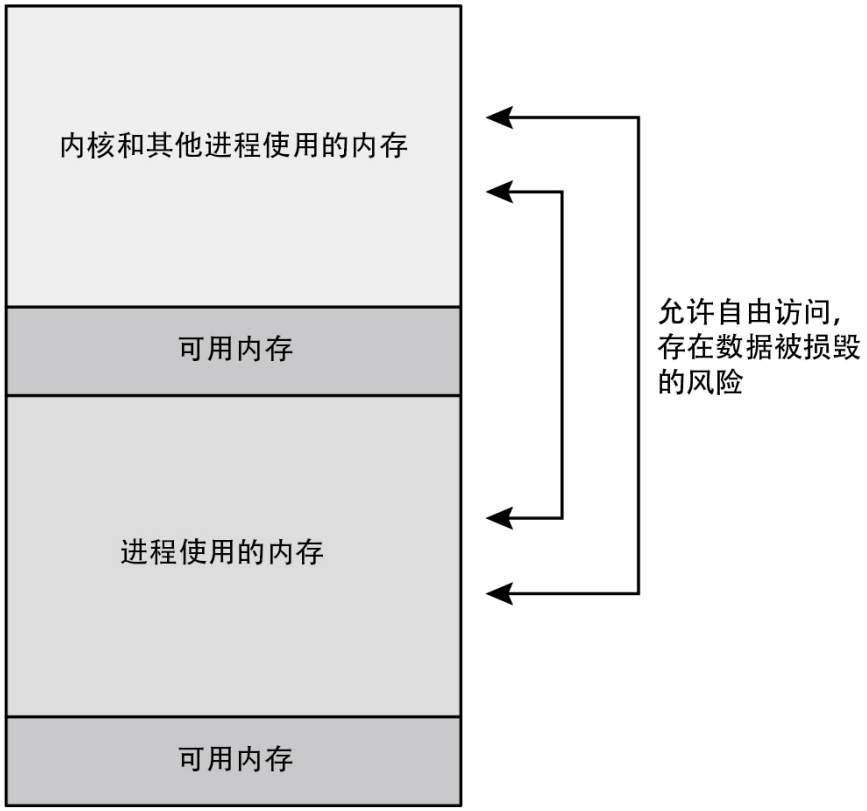

图 5-9　能访问其他已被使用的内存

这样就会存在数据被损毁或泄露的风险。特别需要注意的是，假如内核的数据被损毁了，整个系统将无法正常运行。

● 难以执行多任务

下面来思考一下多个进程同时运行的情景。以如图 5-7 所示的状态为初始状态，如果再次启动同一个程序并尝试映射到内存，就会引发问题。因为对于这个程序来说，如果不把代码放在 300 号地址上，把数据放在 400 号地址上，程序就无法正常运行。