```
Thu Dec 28 13:22:52 JST 2017: sleep 3 seconds
Thu Dec 28 13:22:55 JST 2017: start 2nd read
Thu Dec 28 13:22:55 JST 2017: end 2nd read
$
```

在运行该脚本期间，sar -d -p 命令的输出结果如下所示。

```
$ sar -d -p 1
( 略 )
12:36:35     DEV     tps   rd_sec/s   wr_sec/s   avgrq-sz
avgqu-sz   await   svctm      %util
( 略 )
13:22:43     sda    0.00       0.00       0.00       0.00
0.00       0.00    0.00       0.00
( 略 )
13:22:44     sda  123.00     576.00  215040.00    1752.98
0.24       1.92    1.85      22.80        ←①
( 略 )
13:22:45     sda  446.00    1920.00  790528.00    1776.79
0.82       1.83    1.78      79.00
( 略 )
13:22:46     sda  456.00    3488.00  710656.00    1566.11
0.81       1.77    1.67      76.00
( 略 )
13:22:47     sda  207.00     672.00  380928.00    1843.48
0.39       1.89    1.86      38.40        ←②
( 略 )
13:22:48     sda    0.00       0.00       0.00       0.00
0.00       0.00    0.00       0.00
( 略 )
13:22:49     sda    0.00       0.00       0.00       0.00
0.00       0.00    0.00       0.00
( 略 )
13:22:50     sda  296.00  534528.00       0.00    1805.84
6.85      22.72    1.72      50.80        ←③
( 略 )
13:22:51     sda  577.00 1050624.00       0.00    1820.84
13.64     23.63    1.73     100.00
( 略 )
13:22:52     sda  282.00  512000.00       0.00    1815.60
6.41      23.32    1.72      48.40        ←④
( 略 )
13:22:53     sda    0.00       0.00       0.00       0.00
0.00       0.00    0.00       0.00
( 略 )
13:22:54     sda    0.00       0.00       0.00       0.00
0.00       0.00    0.00       0.00
```

```
( 略 )
13:22:55      sda    0.00         0.00         0.00         0.00
0.00         0.00    0.00         0.00          ←⑤
( 略 )
13:22:56      sda    0.00         0.00         0.00         0.00
0.00         0.00    0.00         0.00          ←⑥
( 略 )
13:22:57      sda    0.00         0.00         0.00         0.00
0.00         0.00    0.00         0.00
( 略 )
```

在 sar -d -p 命令的输出中，rd_sec/s 和 wr_sec/s 分别代表外部存储器（在这里是 sda）每秒的读取数据量和写入数据量。这两个数值以名为**扇区**的单元为单位，该单元的大小为 512 字节。

根据上面的输出，可以得出以下结论。

- **在创建文件时（①与②），向外部存储器写入的数据总量为 1 GB**
- **在初次读取文件时（③与④），从外部存储器读取的数据总量为 1 GB**
- **在第 2 次读取文件时（⑤与⑥），并没有从外部存储器读取数据**

顺便一提，%util 指的是在监测周期（在这个例子中为 1 秒）内，访问外部存储器消耗的时间所占的比例。在 sar -P ALL 等命令中也存在具有相似含义的 %iowait，区别在于，这一数值代表“CPU 处于空闲状态，且在该 CPU 上存在正在等待 I/O 的进程”的时间占比。不过，%iowait 容易造成混淆，是一个没什么用处的数值，大家忽略即可。

6.12　写入文件的实验

下面验证一下文件的写入处理，以及在这之后的后台处理。首先，在禁用页面缓存的直写模式下测试写入文件所需的时间，这一写入方式在文件读取实验中也用过。

```
$ rm -f testfile
$ time dd if=devzero of=testfile oflag=direct bs=1M count=1K
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 2.5601 s, 419 MB/s

real    0m2.561s
user    0m0.012s
sys     0m0.492s
$
```

接着，测试正常使用页面缓存时写入文件所消耗的时间。

```
$ rm -f testfile
$ time dd if=devzero of=testfile bs=1M count=1K
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 0.30129 s, 3.6 GB/s

real    0m0.302s
user    0m0.000s
sys     0m0.300s
```

可以看到，写入速度接近原来的 9 倍。这就是页面缓存在写入文件时所能发挥的优势。

● 采集统计信息

与读取实验时一样，采集写入期间的统计信息，为此我们使用如代码清单 6-3 所示的脚本。

代码清单 6-3　write.sh 脚本

```
#!binbash

```

```
rm -f testfile

echo "$(date): start write (file creation)"
dd if=devzero of=testfile bs=1M count=1K
echo "$(date): end write"

rm -f testfile
```

在执行 sar -B 命令采集统计信息的同时运行 write.sh 脚本，运行结果如下所示。

```
$ ./write.sh
Thu Dec 28 14:11:37 JST 2017: start write (file creation)
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 0.297712 s, 3.6 GB/s
Thu Dec 28 14:11:37 JST 2017: end write
$
```

此时，sar -B 命令的输出结果如下所示。

```
$ sar -B 1
( 略)
14:11:33   pgpgin/s  pgpgout/s  fault/s  majflt/s  pgfree/s ↲
pgscank/s pgscand/s  pgsteal/s   %vmeff
14:11:34       0.00       0.00     2.00      0.00      1.00 ↲
0.00           0.00       0.00     0.00
14:11:35       0.00       0.00     0.00      0.00      2.00 ↲
0.00           0.00       0.00     0.00
14:11:36       0.00       0.00     0.00      0.00      1.00 ↲
0.00           0.00       0.00     0.00
14:11:37       0.00       0.00  1027.00      0.00 263477.00 ↲
0.00           0.00       0.00     0.00      ←①
14:11:38       0.00       0.00     0.00      0.00      4.00 ↲
0.00           0.00       0.00     0.00      ←②
( 略 )
$
```

从测试结果可以得知，在写入期间（①与②），并没有发生页面调出。

然后，我们看一下 I/O 吞吐量的统计信息。在执行 sar -d -p 命令采集统计信息的同时运行 write.sh 脚本，运行结果如下所示。

```
$ ./write.sh
Thu Dec 28 14:17:48 JST 2017: start write (file creation)
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 0.296854 s, 3.6 GB/s
Thu Dec 28 14:17:48 JST 2017: end write
$
```

在运行该脚本期间，sar -d -p 命令的输出结果如下所示。

```
$ sar -d -p 1
( 略 )
14:17:42    DEV    tps    rd_sec/s    wr_sec/s   argrq-sz
avgqu-sz  await  svctm       %util
( 略 )
14:17:44    sda   0.00        0.00        0.00       0.00
0.00       0.00   0.00        0.00
( 略 )
14:17:45    sda   0.00        0.00        0.00       0.00
0.00       0.00   0.00        0.00
( 略 )
14:17:46    sda   0.00        0.00        0.00       0.00
0.00       0.00   0.00        0.00
( 略 )
14:17:47    sda   0.00        0.00        0.00       0.00
0.00       0.00   0.00        0.00
( 略 )
14:17:48    sda   0.00        0.00        0.00       0.00
0.00       0.00   0.00        0.00   ←①
( 略 )
14:17:49    sda   1.00        0.00       16.00      16.00
0.00       0.00   0.00        0.00   ←②
( 略 )
14:17:50    sda   0.00        0.00        0.00       0.00
0.00       0.00   0.00        0.00
( 略 )
$
```

通过上面的信息可以得知，在写入期间，在保存着根文件系统的设备上没有发生 I/O 处理。

6.13 调优参数

Linux 提供了很多用于控制页面缓存的调优参数，这里介绍其中的几个。

在 Linux 中，脏页的回写周期可以通过 sysctl 的 vm.dirty_writeback_centisecs 参数更改。由于该参数的单位是不太常见的厘秒（1/100 秒），所以我们需要花点时间来习惯。该参数的默认值为 500，表示每 5 秒执行一次回写。

```
$ sysctl vm.dirty_writeback_centisecs
vm.dirty_writeback_centisecs = 500
$
```

如果把该参数的值设置为 0，则周期性的回写操作将被禁用。由于这样做非常危险，所以如非为了做实验，请不要这样设置。

Linux 中也有当系统内存不足时防止产生剧烈的回写负荷的参数。通过 vm.dirty_backgroud_ratio 参数可以指定一个百分比值，当脏页占用的内存量与系统搭载的内存总量的比值超过这一百分比值时，后台就会开始运行回写处理，这个参数的默认值为 10（单位：%）。

```
$ sysctl vm.dirty_background_ratio
vm.dirty_background_ratio = 10
$
```

如果想以字节为单位而非以百分比为单位设置该值，可以使用 vm.dirty_background_bytes 参数。该参数的默认值为 0（0 表示不启用[7]）。

[7]百分比与字节这两个版本的参数不能同时启用。——译者注

```
$ sysctl vm.dirty_background_bytes
vm.dirty_background_bytes = 0
```

当脏页的内存占比超过 vm.dirty_ratio 参数指定的百分比值时，将阻塞进程的写入，直到一定量的脏页完成回写处理。该参数的默认值为 20（单位：%）。

```
$ sysctl vm.dirty_ratio
vm.dirty_ratio = 20
$
```

同样，如果想以字节为单位设置该值，可以使用 vm.dirty_bytes 参数。该参数的默认值为 0（0 表示不启用）。

```
$ sysctl vm.dirty_bytes
vm.dirty_bytes = 0
```

通过优化这些参数，能够让系统不至于在内存不足时出现大量的脏页回写处理。

下面介绍的内容与调优参数稍微不同，是清除系统上的（接近全部的）页面缓存的方法。为了清理页面缓存，需要往名为 *proc*sys/vm/drop_caches 的文件中写入 3。

```
# free
              total        used        free      shared  buff/cache   available
Mem:       32941348      241240    31163976        9664     1536132    32203152
                                              ←有接近 1.5 GB 的 buff/cache
Swap:             0           0           0
# echo 3 >procsys/vm/drop_caches
# free
              total        used        free      shared  buff/cache   available
Mem:       32941348      241084    32442940        9664      257324    32253412
                                 ← buff/cache 几乎没了，只剩下约 251 MB
Swap:             0           0           0
#
```

虽然在现实中没什么机会能用上该功能，但这个功能便于我们确认页面缓存对系统性能的影响。顺便一提，不需要在意为什么写入的值是 3，因为这并不重要。

6.14 总结

只要文件上的数据存在于页面缓存中，访问该文件的速度就能快几个数量级。因此，预先评估系统要访问的文件的大小以及物理内存的容量非常重要。

如果在变更设定或者经过一段时间后，系统性能突然大幅下降，那么有可能是因为页面缓存容不下文件的数据。可以通过调整各种 `sysctl` 参数，来抑制页面缓存回写导致的 I/O 负荷增大。另外，可以通过 `sar -B` 和 `sar -d -p` 获取页面缓存相关的统计信息。

6.15 超线程

如前所述，访问内存的延迟比 CPU 运算所消耗的时间长很多。另外，与 CPU 的运算速度相比，访问缓存的速度也稍微要慢一点。因此，在通过 time 命令显示的计入 user 和 sys 的 CPU 使用时间中，大部分浪费在了等待数据从内存或高速缓存传输至 CPU 的过程上，如图 6-17 所示。

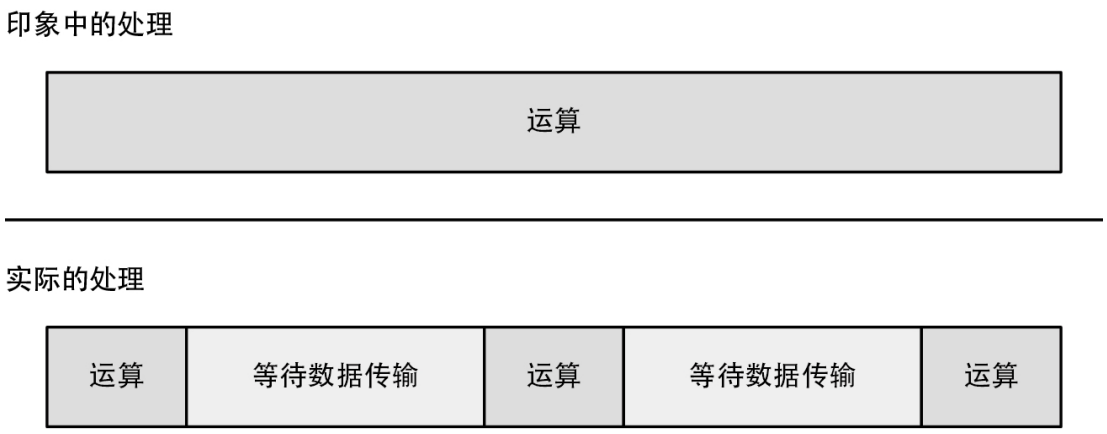

图 6-17 CPU 资源的浪费

top 命令中的 %CPU 或者 sar -P 命令中的 %user 和 %system 等字段显示的 CPU 使用率，同样也包含等待数据传输的时间。通过**超线程**功能，可以有效利用这些浪费在等待上的 CPU 资源[8]。另外，虽然在本书中不会详细说明，但除了数据传输之外，还存在许多浪费 CPU 资源的因素。

[8]注意，超线程与进程上的线程毫无关系。

在使用超线程功能后，可以为 CPU 核心提供多份（一般为两份）硬件资源（其中包含一部分 CPU 核心使用的硬件资源，例如寄存器等），然后将其划分为多个会被系统识别为逻辑 CPU 的超线程。在符合这些特殊条件时，可以同时运行多个超线程。

但超线程带来的并非只有好处，它能产生多大效果很大程度上取决于运行在超线程上的进程的行为。即使在最理想的情况下，吞吐量也不

能翻倍，实际上，能提高 20% 到 30% 已经是非常好的状态了。在某些情况下，吞吐量甚至会下降。

因此，在搭建系统时，有必要实际施加负载，来比较一下超线程启用时和禁用时的性能区别，再确定是否启用该功能。

● 超线程的实验

分别在启用和禁用超线程的情况下，生成超过 2000 万行代码的大型软件——Linux 内核，然后比较一下这两种情况下消耗的时间。内核的生成需要做大量的准备，但受限于篇幅，在此不过多说明。由于本实验难以实施，所以大家参考笔者计算机上的实验结果即可。

● 禁用超线程功能时

此时，系统识别出来的逻辑 CPU 有 8 个，每个逻辑 CPU 分别对应一个 CPU 核心。

```
$ sar -P ALL 1 1
( 略 )
14:39:39    0    0.00    0.00    0.00    0.00    0.00    100.00
14:39:39    1    0.00    0.00    0.00    0.00    0.00    100.00
14:39:39    2    0.00    0.00    0.00    0.00    0.00    100.00
14:39:39    3    0.00    0.00    0.00    0.00    0.00    100.00
14:39:39    4    0.00    0.00    0.00    0.00    0.00    100.00
14:39:39    5    2.00    0.00    0.00    0.00    0.00    98.00
14:39:39    6    0.00    0.00    1.00    0.00    0.00    99.00
14:39:39    7    0.00    0.00    0.00    0.00    0.00    100.00
( 略 )
```

下面确认编译所需的时间。设置编译的并行数为 8，即与 CPU 核心数量相同。由于本实验不需要参考编译过程的日志，所以这里不展示其内容。

```
$ time make -j8 >devnull 2>&1

real    1m33.773s
user    9m35.432s
sys     0m57.140s
```

可以看到，大约消耗了 93.7 秒。

● 启用超线程功能时

首先确认逻辑 CPU 的数量。

```
$ sar -P ALL 1 1
( 略 )
14:46:23     0     2.00    0.00    0.00    0.00    0.00    98.00
14:46:23     1     1.00    0.00    0.00    0.00    0.00    99.00
14:46:23     2     1.98    0.00    0.00    0.00    0.00    98.02
14:46:23     3     2.00    0.00    0.00    0.00    0.00    98.00
14:46:23     4     1.00    0.00    0.00    0.00    0.00    99.00
14:46:23     5     2.00    0.00    0.00    0.00    0.00    98.00
14:46:23     6     1.96    0.00    0.98    0.00    0.00    97.06
14:46:23     7     2.00    0.00    0.00    0.00    0.00    98.00
14:46:23     8     3.00    0.00    1.00    0.00    0.00    96.00
14:46:23     9     1.01    0.00    0.00    0.00    0.00    98.99
14:46:23    10     2.97    0.00    0.00    0.00    0.00    97.03
14:46:23    11     2.00    0.00    0.00    0.00    0.00    98.00
14:46:23    12     1.98    0.00    0.00    0.00    0.00    98.02
14:46:23    13     1.01    0.00    0.00    0.00    0.00    98.99
14:46:23    14     2.00    0.00    0.00    0.00    0.00    98.00
14:46:23    15     1.01    0.00    0.00    0.00    0.00    98.99
( 略 )
```

这次识别到 16 个逻辑 CPU。每个逻辑 CPU 代表的不是一个 CPU 核心，而是 CPU 核心中的一个超线程。只要查看一下 *sys*devices/system/cpu/cpu[CPU 号]/topology/thread_siblings_list，就可以确认哪两个超线程是成对的。这里以 CPU0 为例尝试一下。

```
$ cat sysdevices/system/cpu/cpu0/topology/thread_siblings_list
0-1
$
```

可以看到，逻辑 CPU0 与逻辑 CPU1 为同一个 CPU 核心中的一对超线程。在通过同样的方式确认其他逻辑 CPU 后，可以得知，在笔者的计算机上，2 与 3、4 与 5、6 与 7、8 与 9、10 与 11、12 与 13、14 与 15 这些逻辑 CPU 是成对的。

然后确认编译所需的时间。本次的并行数设置为 16。

```
$ time make -j16 >devnull 2>&1
real    1m13.356s
user    15m2.800s
```

```
sys     1m18.588s
$
```

在启用超线程后，大约需要 73.3 秒，比禁用超线程时快了约 22%。看来在生成内核时，超线程可以发挥其应有的效用。但是，一定不要认为任何时候都能像本实验这样顺利地发挥出超线程的优势。正如之前所说，也存在超线程导致性能下降的情况，因此推荐大家在决定是否启用超线程前，先分别测试一下启用时与禁用时的性能差距，确认到底怎么做更有利于系统运行。

第 7 章　文件系统

Linux 在访问外部存储器中的数据时，通常不会直接访问，而是通过更加便捷的方式——文件系统来进行访问。

也许有人认为在计算机系统上存在文件系统是理所当然的，但可能也有人理解不了文件系统存在的意义。因此，这里我们先考虑一下没有文件系统时的情形，然后以此来解释文件系统的重要性。

外部存储器的功能，说白了就只是“将规定大小的数据写入外部存储器中指定的地址上”而已。假设存在一个容量为 100 GB 的外部存储器，然后将 10 GB 大小的内存区域写入 50 GB 的地址上，如图 7-1 所示。

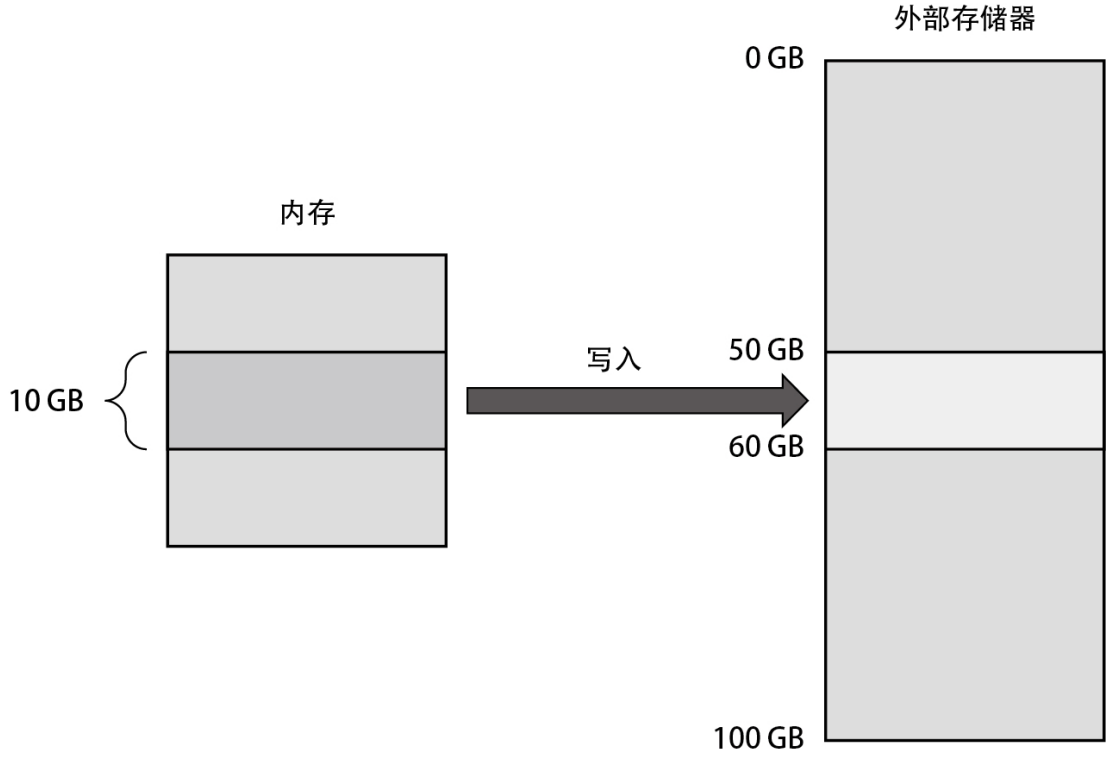

图 7-1　外部存储器的功能

假设大家使用 LibreOffice 之类的办公套件编写了文档，需要把内存中的文档数据保存到外部存储器。此时，如果没有文件系统，就不得不像前面那样自己发出“把 10 GB 大小的数据[1]写入 50 GB 的地址上”的写入请求。

[1]通常不会存在 10 GB 大小的文档数据，这里只是举个例子，大家无须在意这些细节。

在成功保存数据后，还需要亲自记住数据保存的地址、大小以及这份数据的用处，不然下一次就无法读取这份数据了。下面再思考一下需要保存多份数据时的场景。这时，不但要亲自记录所有数据的相关信息，还需要自己管理可用区域，以便掌握设备上的空余容量与空余位置，如图 7-2 所示。

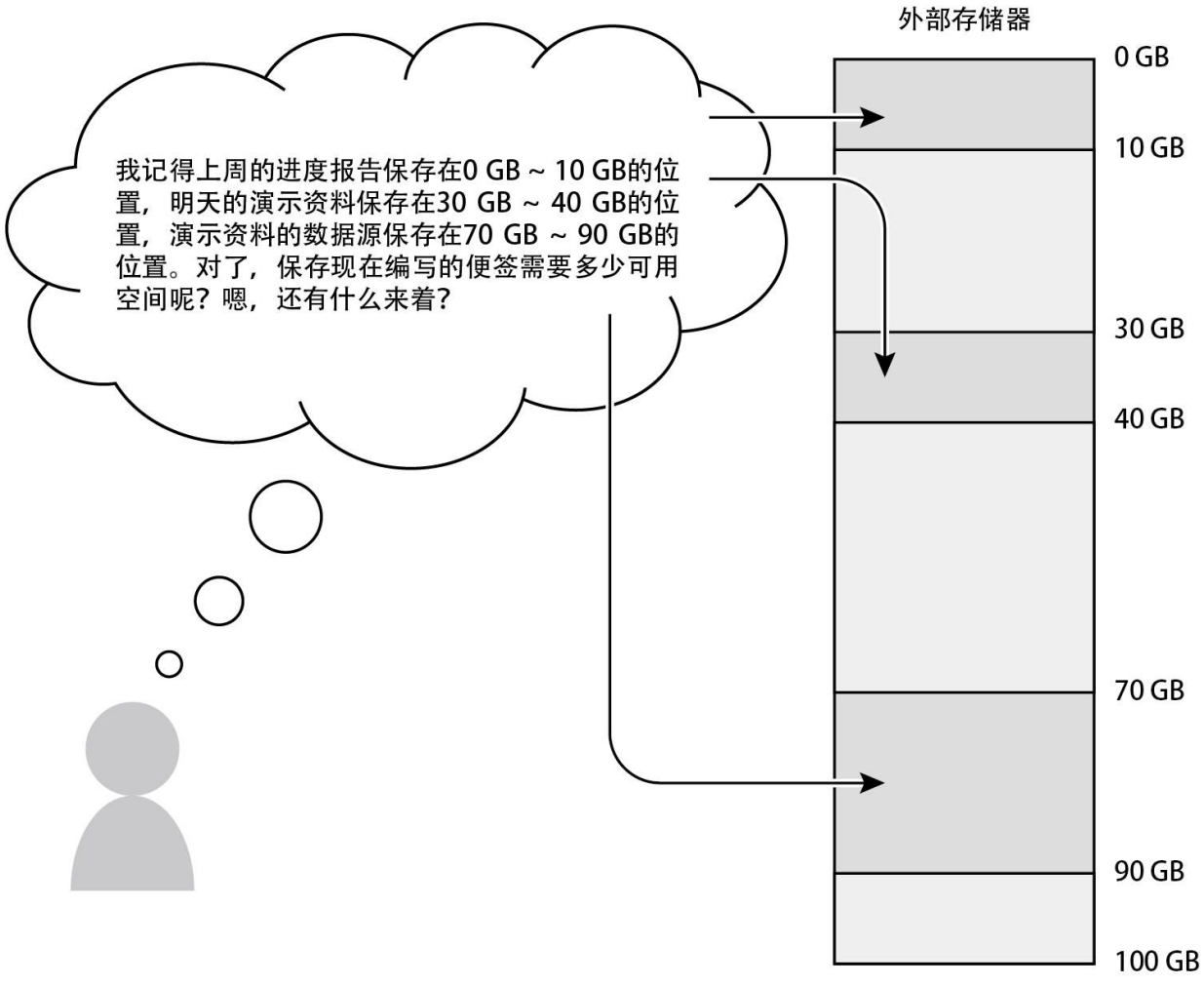

图 7-2　必须管理好所有数据的保存地址以及数据大小等信息

而使用管理着数据保存信息和可用区域等的**文件系统**，即可避免这些繁杂的处理。

文件系统以文件为单位管理所有对用户有实际意义的数据块，并为这些数据块添加上名称、位置和大小等辅助信息。它还规范了数据结构，以确定什么文件应该保存到什么位置，内核中的文件系统将依据该规范处理数据。多亏了文件系统的存在，用户不再需要记住所有数据在外部存储器中的位置与大小等繁杂的信息，只需要记住数据（即文件）的名称即可。

图 7-3 展示了一个非常简单的文件系统，其规格如下所示。

- 文件清单从 0 GB 的位置开始记录
- 记录每个文件的名称、位置和大小这 3 条信息