6.7　转译后备缓冲区

进程需要通过下述步骤访问虚拟地址上的特定数据。

① 对照物理内存中的页表，把虚拟地址转换为物理地址。

② 访问通过步骤①得到的物理地址。

敏锐的人可能已经发现了，这里能发挥高速缓存优势的只有步骤②。这是因为步骤①依然需要访问物理内存，以读取物理内存中的页表。这样一来，特意准备的高速缓存就浪费了。

为了解决这一问题，在 CPU 上存在一个具有与高速缓存同样的访问速度的区域，名为**转译后备缓冲区**（Translation Lookaside Buffer，TLB），又称为快表或页表缓冲，该区域用于保存虚拟地址与物理地址的转换表。利用这一区域，即可提高步骤①的执行速度。这里虽然不会介绍 TLB 的详细内容，但希望大家至少能记住它的名称以及这里关于它的概述。

6.8 页面缓存

与 CPU 访问内存的速度比起来，访问外部存储器的速度慢了几个数量级。内核中用于填补这一速度差的机构称为**页面缓存**。下面将介绍页面缓存的机制及其注意事项。

页面缓存和高速缓存非常相似。高速缓存是把内存上的数据缓存到高速缓存上，而页面缓存则是将外部存储器上的文件数据缓存到内存上。高速缓存以缓存块为单位处理数据，而页面缓存则以页为单位处理数据。

接下来，让我们看一下页面缓存的具体运作流程。

当进程读取文件的数据时，内核并不会直接把文件数据复制到进程的内存中，而是先把数据复制到位于内核的内存上的页面缓存区域，然后再把这些数据复制到进程的内存中，如图 6-12 所示。需要注意的是，方便起见，本节的图中省略了进程的虚拟地址空间，仅画出了物理内存。

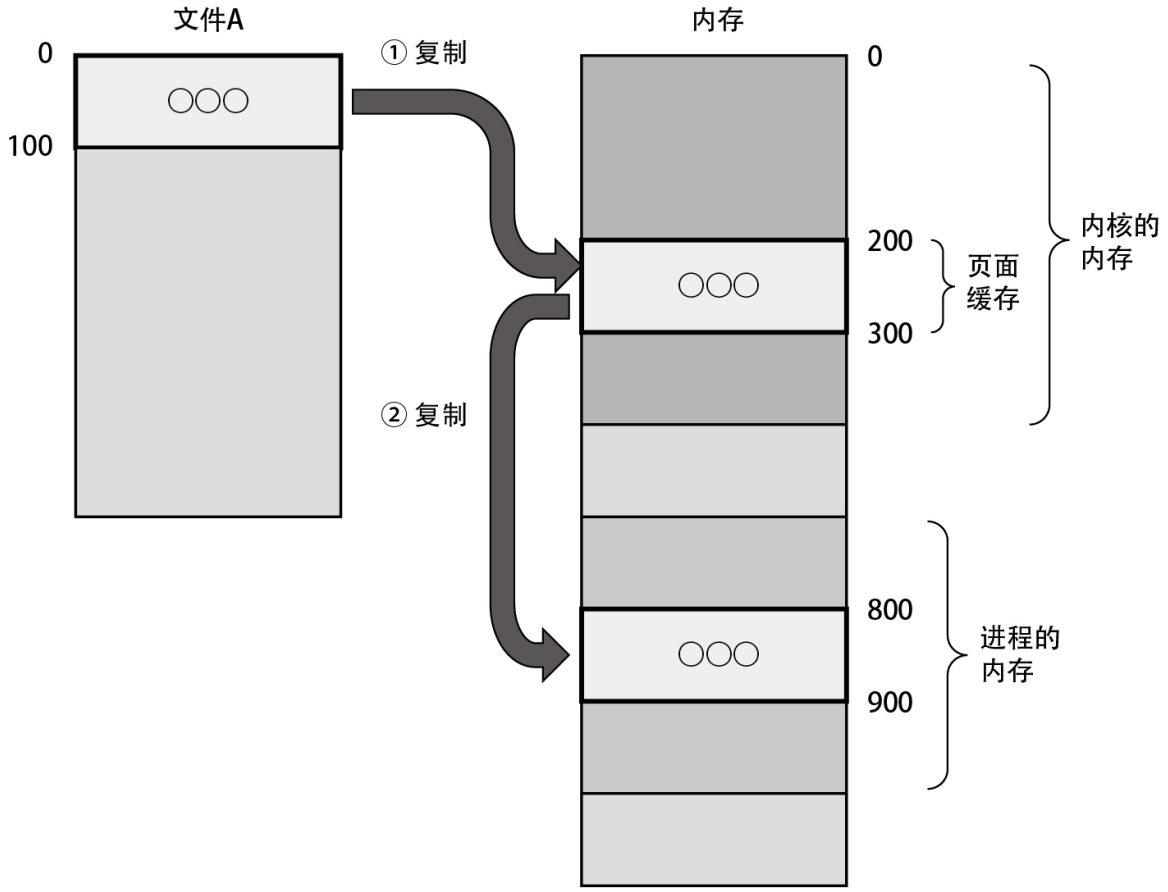

图 6-12　先复制到页面缓存上，然后再复制到进程的内存中

在内核自身的内存中有一个管理区域，该区域中保存着页面缓存所缓存的文件以及这些文件的范围信息等，如图 6-13 所示。

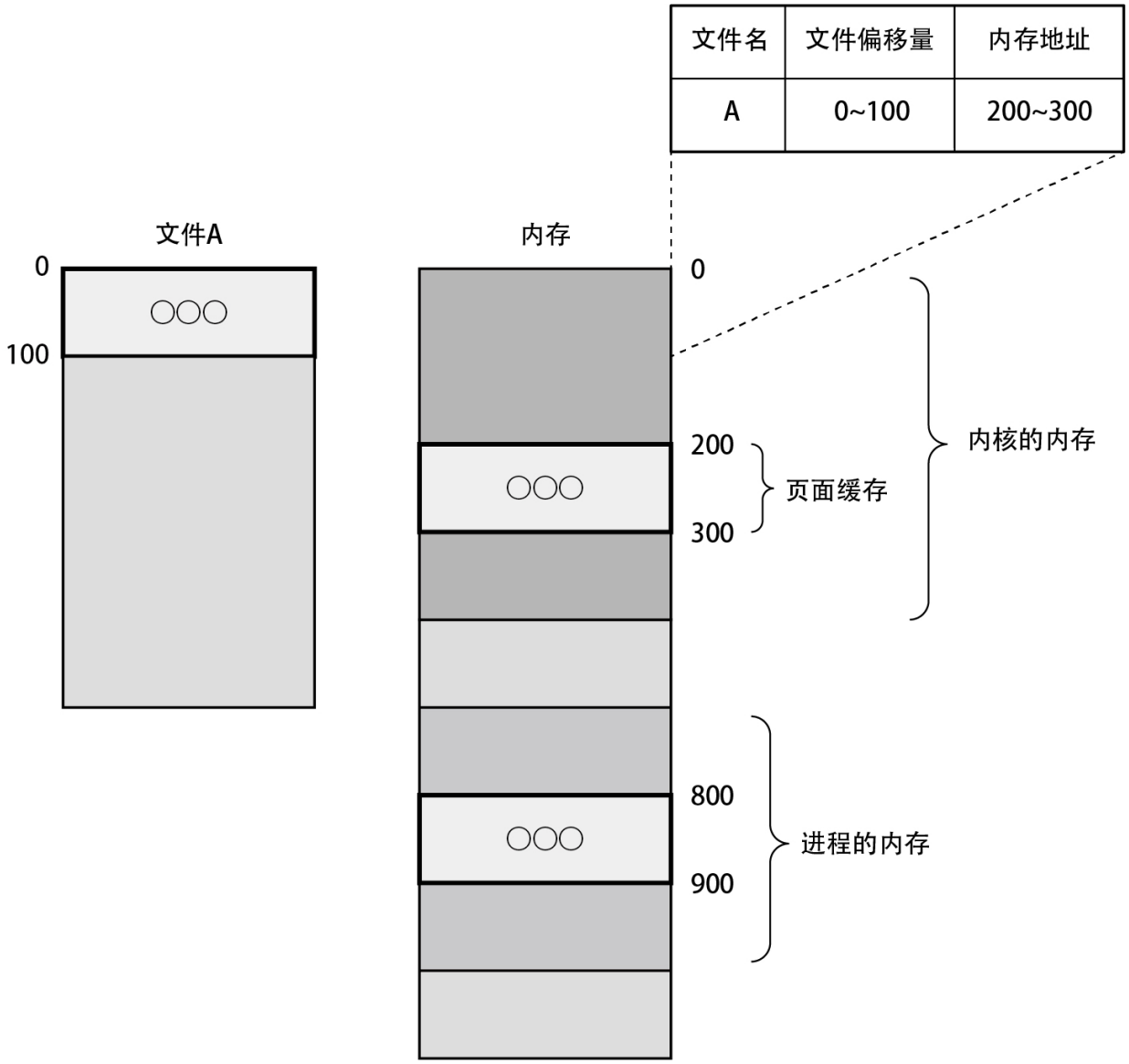

图 6-13　在管理区域中保存着页面缓存所缓存的文件的相关信息

在这之后，如果再次读取已经位于页面缓存的数据，内核将直接返回页面缓存中的数据，如图 6-14 所示。

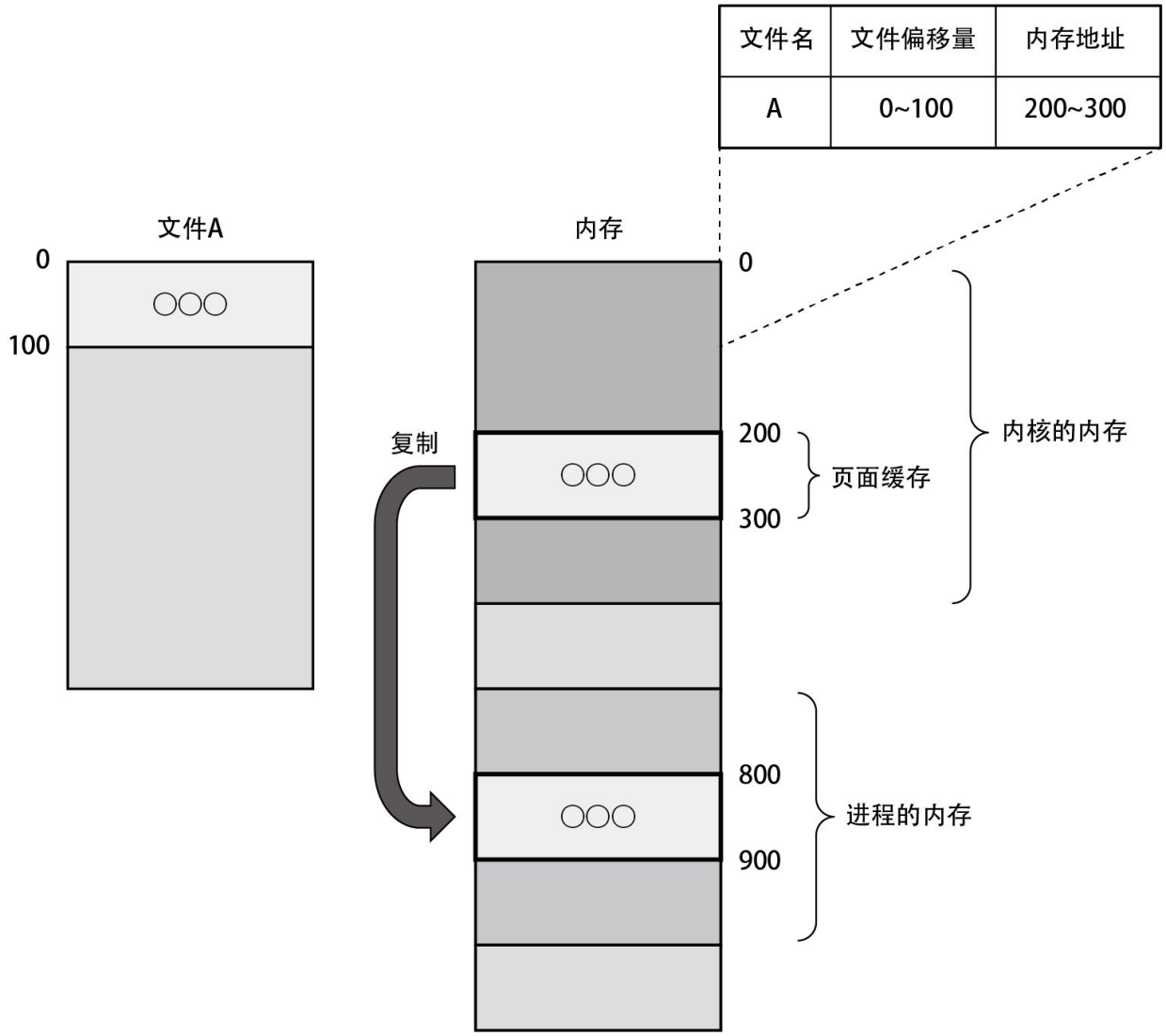

图 6-14　当再次读取同样的数据时，将直接返回页面缓存中的数据

与直接从外部存储器读取相比，从页面缓存读取的速度更快。而且，由于页面缓存是由全部进程共享的资源，所以发起读取的进程也可以不是最初访问该文件数据的进程。

接下来思考一下写入时的情况。

在进程向文件写入数据后，内核会把数据写入页面缓存中，如图 6-15 所示。这时，管理区域中与这部分数据对应的条目会被添加一个标记，以表明“这些是脏数据，其内容比外部存储器中的数据新”。这些被标记的页面称为**脏页**。

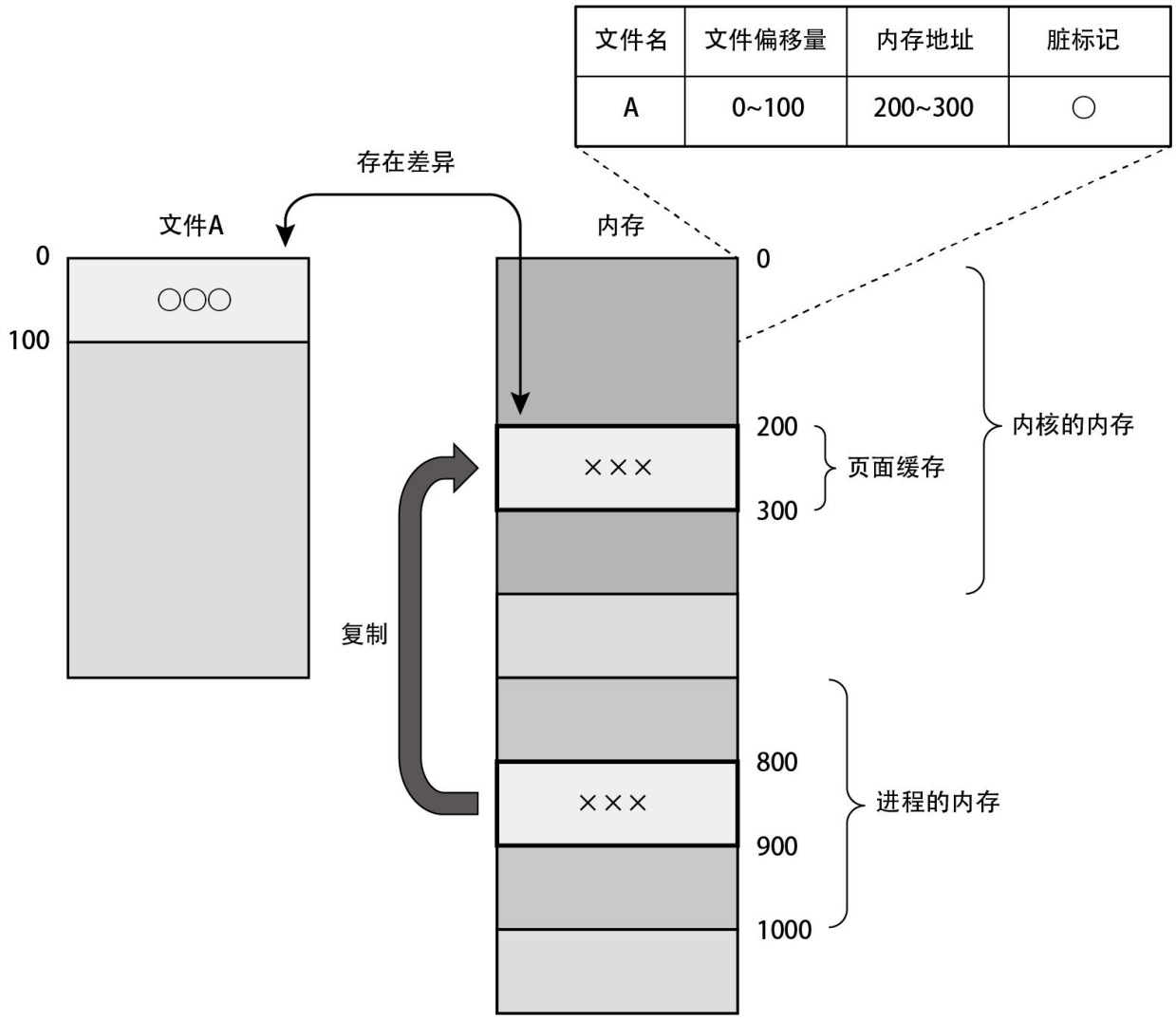

图 6-15　写入操作首先会在页面缓存中执行

与读取操作时一样，比起直接写入外部存储器，写入页面缓存的速度更快。

之后，脏页中的数据将在指定时间通过内核的后台处理反映到外部存储器上。与此同时，脏标记也会被去除，如图 6-16 所示。

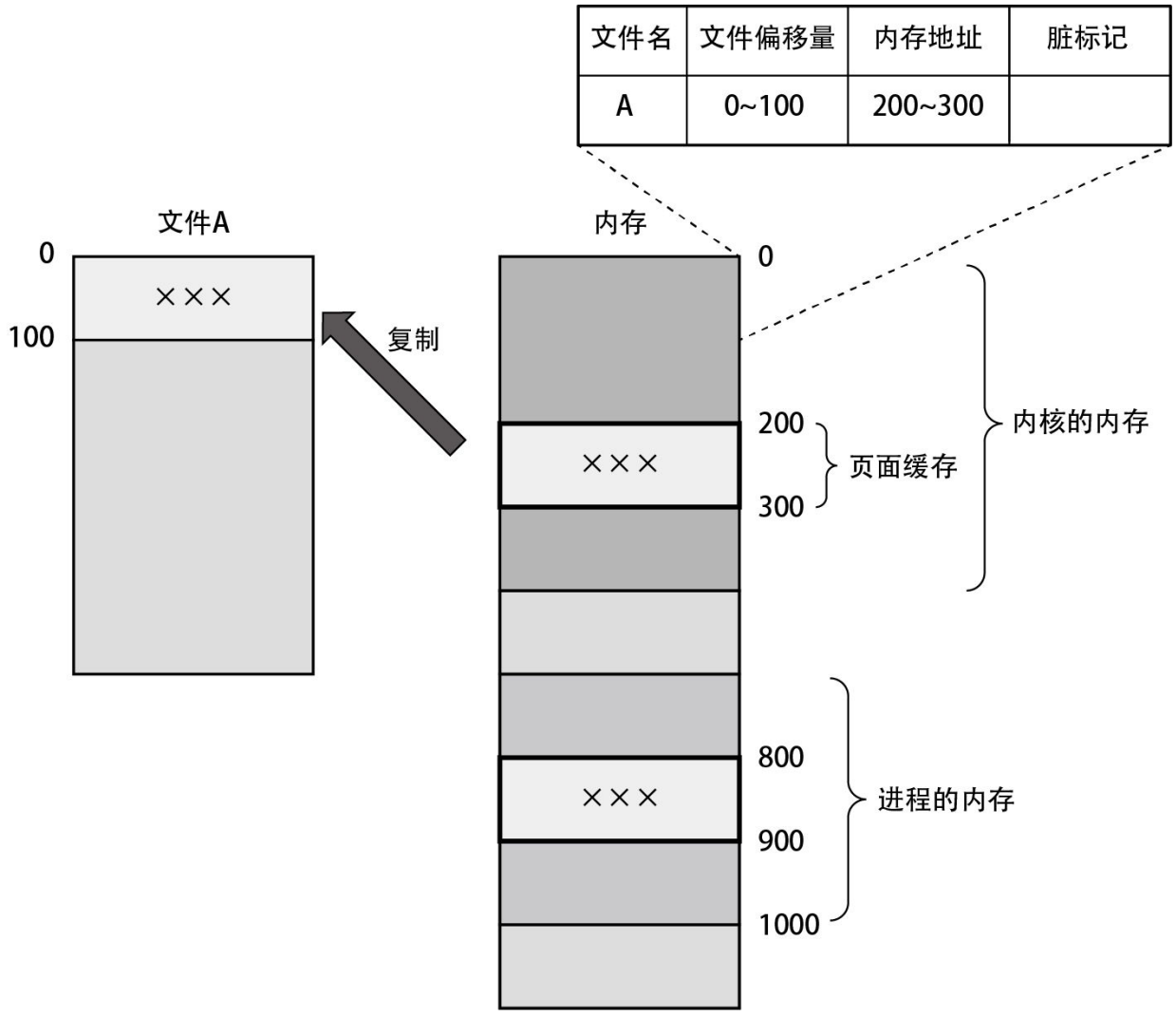

图 6-16　通过后台处理将脏页的数据回写到外部存储器

如果各个进程想要访问的文件数据都已存在于页面缓存中，那么系统的文件访问速度将能超越外部存储器的访问速度，接近内存的访问速度，因此可以期待系统整体运行速度的提升。

需要注意的是，只要系统上还存在可用内存，则每当各个进程访问那些尚未读取到页面缓存中的文件时，页面缓存的大小就会随之增大。

而当系统内存不足时，内核将释放页面缓存以空出可用内存。此时，首先丢弃脏页以外的页面。如果还是无法空出足够内存，就对脏页执行回写，然后继续释放页面。当需要释放脏页时，由于需要访问外部存储器，所以恐怕会导致系统性能下降。尤其是当系统上存在大量文

件写入操作而导致出现大量脏页时，系统负载往往会变得非常大。内存不足引发大量脏页的回写处理，进而导致系统性能下降的情况非常常见。

6.9 同步写入

在页面缓存中还存在脏页的状态下，如果系统出现了强制断电的情况，会发生什么呢？

强制断电将导致页面缓存中的脏页丢失。如果不希望文件访问出现这种情况，请在利用 `open()` 系统调用打开文件时将 `flag` 参数设定为 `O_SYNC`。这样一来，之后每当对该文件执行 `write()` 系统调用，都会在往页面缓存写入数据时，将数据同步写入外部存储器。

6.10　缓冲区缓存

缓冲区缓存是与页面缓存相似的机制。这是当跳过文件系统，通过设备文件直接访问外部存储器时使用的区域。关于设备文件的内容，我们将在下一章介绍。大致上，页面缓存与缓冲区缓存可以概括为“用于将外部存储器中的数据放到内存上的机制”。

6.11 读取文件的实验

为了验证页面缓存的效果，接下来将对同一个文件执行两次读取操作，并对比两次处理所消耗的时间。

首先需要创建一个用于实验的文件。这里将创建一个名为 testfile 的文件，文件大小为 1 GB。

```
$ 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, 2.74668 s, 391 MB/s
```

这里添加了参数 oflag=direct，表示通过直写的方式写入文件。在本次写入操作中，不会使用页面缓存。也就是说，此时在页面缓存中并不存在 testfile 文件。

接下来开始读取 testfile 文件。在测试前后分别采集一次系统中的页面缓存的使用量信息。

```
$ free
              total        used        free      shared  buff/cache   available
Mem:       32941348      203820    32441740        9664      295788    32272416
Swap:             0           0           0
$ time cat testfile >devnull

real    0m2.002s
user    0m0.000s
sys     0m0.468s
$free

              total        used        free      shared  buff/cache   available
Mem:       32941348      205336    31385744        9664     1350268    32244916
Swap:             0           0           0
$
```

可以看到，读取文件大约需要 2 秒。因为这是首次读取，所以本次读取访问的是外部存储器。由于进程本身或响应进程请求的内核使用 CPU 的时间共 0.468 秒，所以可以得知，等待外部存储器的读取操作占用了总体约 3/4 的时间，也就是约 1.54 秒。另外，也可以通过上

面的数据得知，执行测试后的页面缓存比测试前增加了约 1 GB。增加的这一部分就是 testfile 文件的页面缓存。

接着执行第 2 次读取操作。在读取后，再次确认页面缓存的使用量信息。

```
$ time cat testfile >devnull

real    0m0.100s
user    0m0.000s
sys     0m0.104s
$free
              total        used        free      shared  buff/cache   available
Mem:       32941348      205036    31385760        9664     1350552    32245312
Swap:             0           0           0
```

第 2 次的访问速度约为第 1 次的 20 倍，因为第 2 次只是复制页面缓存中的数据而已，不需要访问外部存储器中的数据。需要注意的是，由于 testfile 文件已经位于页面缓存中，所以系统中的页面缓存总量并不会发生变化。

另外，页面缓存的总量不但能通过 free 命令得到，还能通过 sar -r 命令中的 kbcached 字段获知（单位：KB）。如果需要每隔一定的时间观测一次数据，使用后者更加方便。

```
$ 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
```

在实验结束后，记得删除文件。

```
$ rm testfile
$
```

● 采集统计信息

接下来，我们来确认一下在执行上述操作时系统上的统计信息。这里将采集以下 3 种信息。

- 在从外部存储器往页面缓存读入数据时，总共执行了多少次页面调入？
- 在从页面缓存往外部存储器写入数据时，总共执行了多少次页面调出？
- 外部存储器的 I/O 吞吐量是多少？

简单起见，这里把需要执行的所有操作汇集到如代码清单 6-2 所示的脚本中。

代码清单 6-2　read-twice 脚本（read-twice.sh）

```
#!/bin/bash

rm -f testfile

echo "$(date): start file creation"
dd if=/dev/zero of=testfile oflag=direct bs=1M count=1K
echo "$(date): end file creation"

echo "$(date): sleep 3 seconds"
sleep 3

echo "$(date): start 1st read"
cat testfile >/dev/null

echo "$(date): end 1st read"

echo "$(date): sleep 3 seconds"
sleep 3

echo "$(date): start 2nd read"
cat testfile >/dev/null

echo "$(date): end 2nd read"

rm -f testfile
```

首先采集页面调入与页面调出的相关信息。这部分信息可以通过 sar -B 命令采集。在开始采集时，一边运行 read-twice.sh 脚本，一边在后台运行 sar -B 命令。read-twice.sh 脚本的运行结果如下所示。

```
$ ./read-twice.sh
Thu Dec 28 13:04:04 JST 2017: start file creation
1024+0 records in
```

```
1024+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 2.98329 s, 360MB/s
Thu Dec 28 13:04:07 JST 2017: end file creation
Thu Dec 28 13:04:07 JST 2017: sleep 3 seconds
Thu Dec 28 13:04:10 JST 2017: start 1st read
Thu Dec 28 13:04:12 JST 2017: end 1st read
Thu Dec 28 13:04:12 JST 2017: sleep 3 seconds
Thu Dec 28 13:04:15 JST 2017: start 2nd read
Thu Dec 28 13:04:16 JST 2017: end 2nd read
```

另一个终端上的 sar -B 命令的输出如下所示。

```
$ sar -B 1
( 略 )
13:03:42   pgpgin/s  pgpgout/s  fault/s majflt/s  pgfree/s ↲
pgscank/s pgscand/s  pgsteal/s   %vmeff
( 略 )
13:04:02      0.00       0.00      0.00     0.00      2.00 ↲
0.00          0.00       0.00      0.00
13:04:03      0.00       0.00      0.00     0.00      2.00 ↲
0.00          0.00       0.00      0.00
13:04:04      0.00       0.00      0.00     0.00      2.00 ↲
0.00          0.00       0.00      0.00
13:04:05    256.00  206848.00    749.00     0.00    240.00 ↲
0.00          0.00       0.00      0.00     ←①
13:04:06   1552.00  372736.00      0.00     0.00      3.00 ↲
0.00          0.00       0.00      0.00
13:04:07   1216.00  331776.00      0.00     0.00      1.00 ↲
0.00          0.00       0.00      0.00
13:04:08    416.00  137216.00    363.00     0.00    506.00 ↲
0.00          0.00       0.00      0.00     ←②
13:04:09      0.00       0.00      0.00     0.00      3.00 ↲
0.00          0.00       0.00      0.00
13:04:10      0.00       0.00      0.00     0.00      2.00 ↲
0.00          0.00       0.00      0.00
13:04:11 286208.00       0.00    275.00     0.00    212.00 ↲
0.00          0.00       0.00      0.00     ←③
13:04:12 524288.00       0.00      0.00     0.00    106.00 ↲
0.00          0.00       0.00      0.00
13:04:13 238080.00       0.00    361.00     0.00    288.00 ↲
0.00          0.00       0.00      0.00     ←④
13:04:14   3312.00   24252.00      0.00     0.00    890.00 ↲
0.00          0.00       0.00      0.00
13:04:15      0.00       0.00      0.00     0.00      1.00 ↲
0.00          0.00       0.00      0.00     ←⑤
13:04:16    112.00       0.00    538.00     0.00 263182.00 ↲
0.00          0.00       0.00      0.00     ←⑥
13:04:17      0.00       0.00      0.00     0.00      1.00 ↲
```

```
0.00            0.00        0.00        0.00
13:04:18        0.00        0.00        0.00      0.00        2.00 ↵
0.00            0.00        0.00        0.00
( 略 )
$
```

通过对照两个终端输出的内容中的时间戳，比较不同时间点的输出内容，可以得出以下结论。

- **在创建文件时（①与②），调出的页面总量为 1 GB[6]**
- **在初次读取文件时（③与④），调入的页面总量为 1 GB。此时从外部存储器往页面缓存读取了数据**
- **在第 2 次读取文件时（⑤与⑥），并没有发生页面调入。页面调入字段的细微变化是由系统上的其他进程引起的**

[6]这里容易令人困惑的是，即便禁用页面缓存，直接往外部存储器写入文件数据，这部分数据流量也会被计入页面调出字段的数值中。

接下来，确认一下外部存储器的 I/O 吞吐量。这次使用的是 sar -d -p 命令。利用该命令，可以显示每个外部存储器的 I/O 相关的信息。首先，需要确认作为写入目标的文件系统的名称，即获取根文件系统所在的外部存储器的名称。

```
$ mount | grep "on / "
devsda5 on  type btrfs (rw,relatime,ssd,space_cache, ↵
subvolid=257,subvol=@)
$
```

可以看到，在笔者的计算机上，该设备的名称为 *dev*sda5，它代表名为 sda 的外部存储器的第 5 个分区。借助 sar -d -p 命令，我们可以监控 sda 上的数据。下面我们开始进行测试，一边执行 sar -d -p 命令，一边运行 read-twice.sh 脚本，运行结果如下所示。

```
$ ./read-twice.sh
Thu Dec 28 13:22:44 JST 2017: start file creation
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 2.81054 s, 382MB/s
Thu Dec 28 13:22:47 JST 2017: end file creation
Thu Dec 28 13:22:47 JST 2017: sleep 3 seconds
Thu Dec 28 13:22:50 JST 2017: start 1st read
Thu Dec 28 13:22:52 JST 2017: end 1st read
```