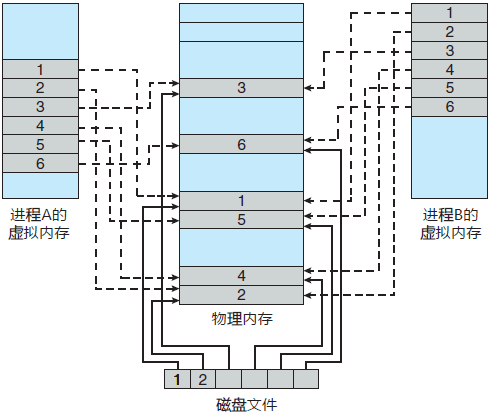
# 内存映射文件完全攻略

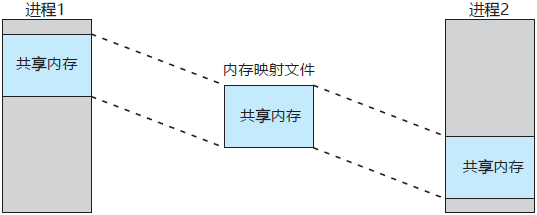
假设采用标准系统调用 open()、read() 和 write() 来顺序读取磁盘文件，每个文件访问都需要系统调用和磁盘访问。又或者采用虚拟内存技术，以将文件 I/O 作为常规内存访问，这种方法称为内存映射文件，允许一部分虚拟内存与文件进行逻辑关联，这会导致显著的性能提高。

**基本机制**

实现文件的内存映射是将每个磁盘块映射到一个或多个内存页面。  
  
最初，文件访问按普通请求调页来进行，从而产生缺页错误。这样，文件的页面大小部分从文件系统读取到物理页面（有些系统可以选择一次读取多个页面大小的数据块）。以后，文件的读写就按常规内存访问来处理。通过内存的文件操作，没有采用系统调用 read() 和 write() 的开销，而且简化了文件的访问和使用。  
  
请注意，内存映射文件的写入不一定是对磁盘文件的即时（同步）写入。有的操作系统定期检查文件的内存映射页面是否已被修改，以便选择是否更新到物理文件。当关闭文件时，所有内存映射的数据会写到磁盘，并从进程虚拟内存中删除。  
  
有些操作系统仅通过特定的系统调用来提供内存映射，而通过标准的系统调用来处理所有其他文件 I/O。然而，有的系统不管文件是否指定为内存映射，都选择对文件进行内存映射。  
  
以 Solaris 为例，如果一个文件指定为内存映射（采用系统调用 mmaP()），那么 Solaris 会将该文件映射到进程地址空间。如果一个文件通过普通系统调用，如 open()、read() 和 write() 来打开和访问，那么 Solaris 仍然采用内存映射文件。然而，这个文件是映射到内核地址空间，无论文件如何打开，Solaris 都将所有文件 I/O 视为内存映射的，以允许文件访问在高效的内存子系统中进行。  
  
多个进程可以允许并发地内存映射同一文件，以便允许数据共享。任何一个进程的写入会修改虚拟内存的数据，并且其他映射同一文件部分的进程都可看到。  
  
根据虚拟内存的相关知识，可以清楚地看到内存映射部分的共享是如何实现的：每个共享进程的虚拟内存映射指向物理内存的同一页面，而该页面有磁盘块的复制，这种内存共享如图 1 所示。

  
图 1 内存映射文件

内存映射系统调用还可以支持写时复制功能，允许进程既可以按只读模式来共享文件，也可以拥有自己修改的任何数据的副本。为了协调对共享数据的访问，有关进程可以使用实现互斥的机制。  
  
很多时候，共享内存实际上是通过内存映射来实现的。在这种情况下，进程可以通过共享内存来通信，而共享内存是通过映射同样文件到通信进程的虚拟地址空间来实现的。内存映射文件充当通信进程之间的共享内存区域（图 2）。

  
图 2 采用内存映射 I/O 的共享内存

首先创建 POSIX 共享内存对象，然后每个通信进程内存对象映射到其地址空间。在接下来的部分，将说明 Windows API 如何支持通过内存映射文件的内存共享。

**共享内存 Windows API**

通过内存映射文件的 Windows API 以创建共享内存区域的大致过程是这样的，首先为要映射的文件创建文件映射，接着在进程虚拟地址空间中建立映射文件的视图。另一个进程可以打开映射的文件，并且在虚拟地址空间中创建它的视图。映射文件表示共享内存对象，以便进程能够通信。  
  
接下来更详细地说明这些步骤。首先，生产者进程使用 Windows API 中的内存映射功能来创建共享内存对象。接着，生产者将消息写入共享内存。然后，生产者进程打开对共享内存对象的映射，并读取生产者写入的消息。  
  
为了建立内存映射文件，进程首先通过函数 CreateFile() 打开需要映射的文件，并得到打开文件的 HAIJDLE（句柄）。接着，进程通过函数 CreateFileMapping() 创建这个文件的映射。  
  
一旦建立了文件映射，进程然后通过函数 MapViewOfFile()，在虚拟地址空间中建立映射文件的视图。映射文件的视图表示位于进程虚拟地址空间中的映射文件的部分，可以是整个文件或者是映射文件的一部分。如下所示程序说明了这个顺序。（为使代码简洁，这里省略了大量的错误检查。）

1. #include <windows.h>
2. #include <stdio.h>
3. int main(int argc, char \*argv[])
4. {
5. HANDLE hFile, hMapFile;
6. LPVOID lpMapAddress;
7. hFile = CreateFile("temp.txt", /\* file name \*/
8. GENERIC\_READ I GENERIC\_WRITE, /\* read/write access \*/
9. 0, /\* no sharing of the file \*/
10. NULL, /\* default security \*/
11. 0PEN\_ALWAYS, /\* open new or existing file \*/
12. FILE\_ATTRIBUTE\_NORMAL, /\* routine file attributes \*/
13. NULL); /\* no file template \*/
14. hMapFile = CreateFileMapping(hFile, /\* file handle \*/
15. NULL, /\* default security \*/
16. PAGE\_READWRITE, /\* read/write access to mapped pages \*/
17. 0， /\* map entire file \*/
18. 0,
19. TEXT("SharedObject"))； /\* named shared memory object \*/
20. lpMapAddress = MapViewOfFile(hMapFile, /\* mapped object handle \*/
21. FILE\_MAP\_ALL\_ACCESS, /\* read/write access \*/
22. 0, /\* mapped view of entire file \*/
23. 0,
24. 0)；
25. /\* write to shared memory \*/
26. sprintf(lpMapAddress,"Shared memory message");
27. UnmapViewOfFile(lpMapAddress);
28. CloseHandle(hFile);
29. CloseHandle(hMapFile);
30. }

调用 CreateFileMapping() 创建一个名为 SharedObject 的命名共享内存对象。消费者进程创建这个命名对象的映射，从而利用这个共享内存段进行通信。  
  
接着，生产者在它的虚拟地址空间中创建内存映射文件的视图。通过将0传递给最后三个参数，表明映射的视图为整个文件。通过传递指定的偏移和大小，这样创建的视图只包含文件的一部分。

注意，在建立映射后，整个映射可能不会加载到内存中。映射文件可能是请求调页的，因此只有在被访问时才将所需页面加载到内存。

函数 MapViewOfFile() 返回共享内存对象的指针，因此，对这个内存位置的任何访问就是对共享内存文件的访问。在这个例子中，生产者进程将消息"Shared memory message"写到共享内存。  
  
如下所示的程序说明了消费者进程如何建立命名共享内存对象的视图。这个程序比之前的程序要简单，因为这个进程所需做的就是创建一个到现有的命名共享内存对象的映射。消费者进程也必须创建映射文件的视图，这与之前程序的生产者进程一样。然后，消费者就从共享内存中读取由生产者进程写入的消息"Shared memory message"。

1. #include <windows.h>
2. #include <stdio.h>
3. int main(int argc, char \*argv[])
4. {
5. HANDLE hMapFile;
6. LPVOID lpMapAddress;
7. hMapFile = OpenFileMapping(FILE\_MAP\_ALL\_ACCESS, /\* R/W access \*/
8. FALSE, /\* no inheritance 氺/
9. TEXT("SharedObj ect"))； /\* name of mapped file object \*/
10. lpMapAddress = MapViewOfFile(hMapFile, /\* mapped object handle \*/
11. FILE\_MAP\_ALL\_ACCESS, /\* read/write access \*/
12. 0, /\* mapped view of entire file \*/
13. 0,
14. 0)；
15. /\* read from shared memory \*/
16. printf ("Read message %s", lpMapAddress);
17. UnmapViewOfFile(lpMapAddress);
18. CloseHandle(hMapFile);
19. }

最后，两个进程调用 UnmapViewOfFile() 来删除映射文件的视图。

**内存映射 I/O**

在 I/O 的情况下，每个 I/O 控制器包括保存命令和传输数据的寄存器。通常，专用 I/O 指令允许在这些寄存器和系统内存之间进行数据传输。为了更方便地访问 I/O 设备，许多计算机体系结构提供了内存映射 I/O。  
  
在这种情况下，一组内存地址专门映射到设备寄存器。对这些内存地址的读取和写入，导致数据传到或取自设备寄存器。这种方法适用于具有快速响应时间的设备，例如视频控制器。对于 IBMPC，屏幕上的每个位置都映射到一个内存位置。在屏幕上显示文本几乎和将文本写入适当内存映射位置一样简单。  
  
内存映射 I/O 也适用于其他设备，如用于联结 modem 和打印机的计算机串口和并口，通过读取和写入这些设备寄存器（称为 I/O 端口），CPU 可以对这些设备传输数据。  
  
为了通过内存映射串行端口发送一长串字节，CPU 将一个数据字节写到数据寄存器，并将控制寄存器的一个位置位以表示有字节可用。设备读取数据字节，并清零控制寄存器的指示位，以表示已准备好接收下一个字节。接着，CPU 可以传输下一个字节。如果 CPU 采用轮询监视控制位，不断循环查看设备是否就绪，这种操作称为程序 I/O。如果 CPU 不是轮询控制位，而是在设备准备接收一个字节时收到中断，则数据传输称为中断驱动。