```
    printf("*** succeeded to allocate memory: address = %p;
           size = 0x%x ***\n", new_memory, ALLOC_SIZE);
    puts("");

    puts("*** memory map after memory allocation ***");
    fflush(stdout);
    system(command);

    if (munmap(new_memory, ALLOC_SIZE) == -1)
        err(EXIT_FAILURE, "munmap() failed");
    exit(EXIT_SUCCESS);
}
```

在这个程序中，mmap() 函数会通过系统调用向 Linux 内核请求新的内存。另外，system() 函数会执行第 1 个参数中指定的命令。本程序利用这个函数输出了申请内存前后的内存映射信息。

运行这个程序，结果如下。

```
$ cc -o mmap mmap.c
$ ./mmap
*** memory map before memory allocation ***
( 略 )
*** succeeded to allocate memory: address = 0x7f06ce1cc000; ↲
                                      size = 0x6400000 ***
                                                          ←①
( 略 )
*** memory map after memory allocation ***
( 略 )
7f06ce1cc000-7f06d45cc000 rw-p 00000000 00:00 0           ←②
( 略)
```

①所指的内容表明成功映射到地址为 0x7f06ce1cc000 的内存。因此，在执行 mmap() 函数后，比执行该函数前多出了②所指的表示内存区域 7f06ce1cc000-7f06d45cc000 的行。这就是新获得的内存区域。第 1 个数值为新内存区域的起始地址，第 2 个数值为结束地址（都以十六进制数表示）。

最后，确认一下这个内存区域的大小。

```
$ python3 -c "print(0x7f06d45cc000 - 0x7f06ce1cc000)"
104857600
$
```

程序准确无误地分配了 104857600 字节，即 100 MB 的内存。需要注意的是，大家在自己的计算机上运行本程序时，应该会出现与上面的例子不一样的起始地址与结束地址，但无须在意，因为每次运行程序，这个值都会发生改变。不过，不管出现什么值，两个地址的差都是 100 MB。

5.9 利用上层进行内存分配

C 语言标准库中存在一个名为 malloc() 的函数，用于获取内存。在 Linux 中，这个函数的底层调用了 mmap() 函数（图 5-20）。

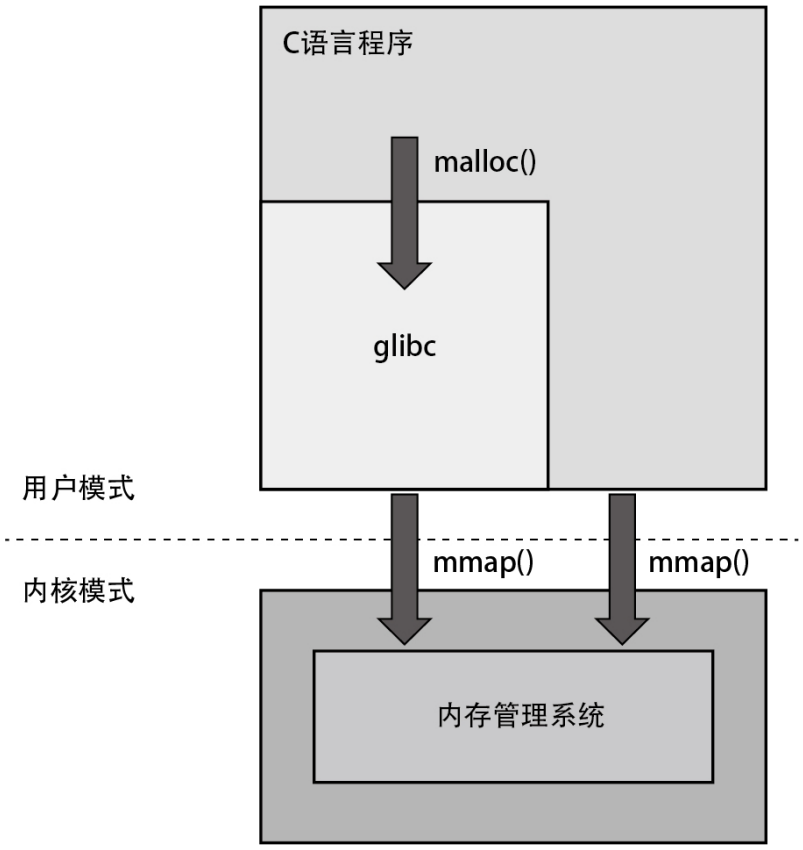

图 5-20 C 语言标准库的 malloc() 函数

mmap() 函数是以页为单位获取内存的，而 malloc() 函数是以字节为单位获取内存的。为了以字节为单位获取内存，glibc 事先通过系统调用 mmap() 向内核请求一大块内存区域作为内存池，当程序调用 malloc() 函数时，从内存池中根据申请的内存量划分出相应大小（以字节为单位）的内存并返回给程序（图 5-21）。在内存池中的内存消耗完后，glibc 会再次调用 mmap() 以申请新的内存区域。

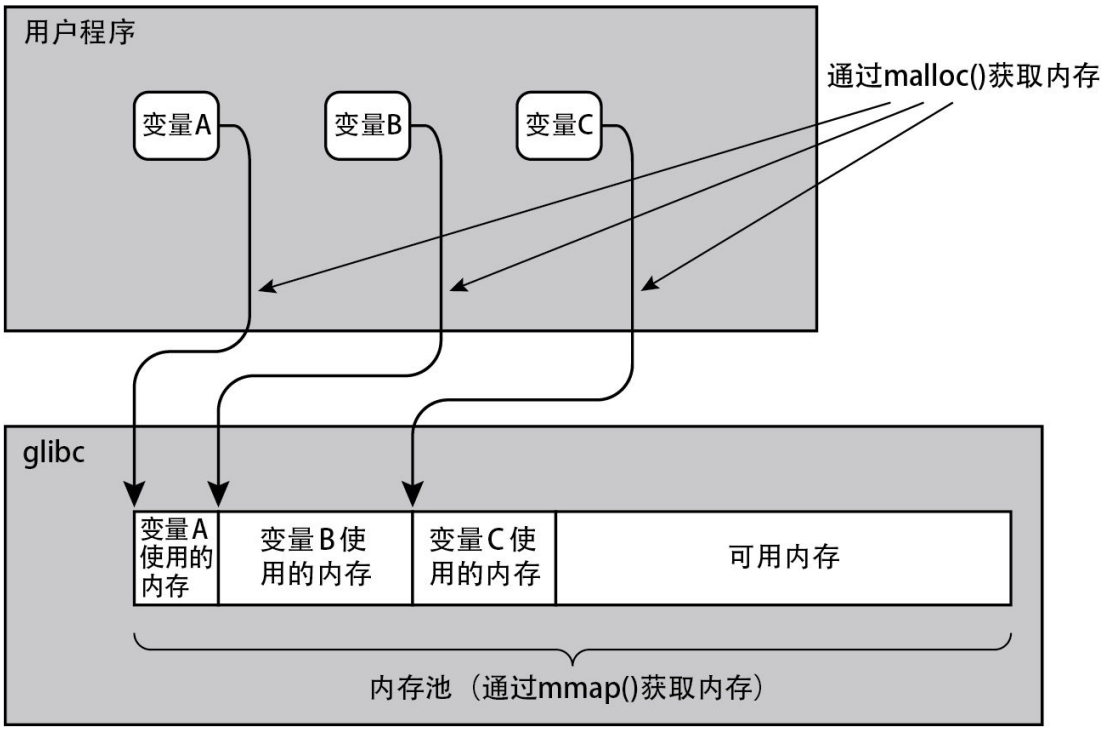

图 5-21　glibc 申请内存形成内存池

这是运行在用户模式下的 OS 功能（glibc 的 malloc() 函数）为普通程序提供的一个典型功能。

顺便一提，虽然部分程序拥有统计自身占用的内存量的功能，但往往程序汇报的值与 Linux 显示的进程的内存消耗量不同，而后者通常会更大。

这是因为，Linux 显示的值包括创建进程时以及调用 mmap() 函数时分配的所有内存，而程序统计的值通常只有通过调用 malloc() 等函数而申请的内存量。如果想知道程序显示的内存消耗量具体统计了哪些数值，可以查看各个程序的说明文档。

另外，即使是 Python 这类将内存管理从源代码上隐藏起来的脚本语言，其项目底层依然是通过 C 语言的 malloc() 函数或 mmap() 函数来获取内存的（图 5-22）。

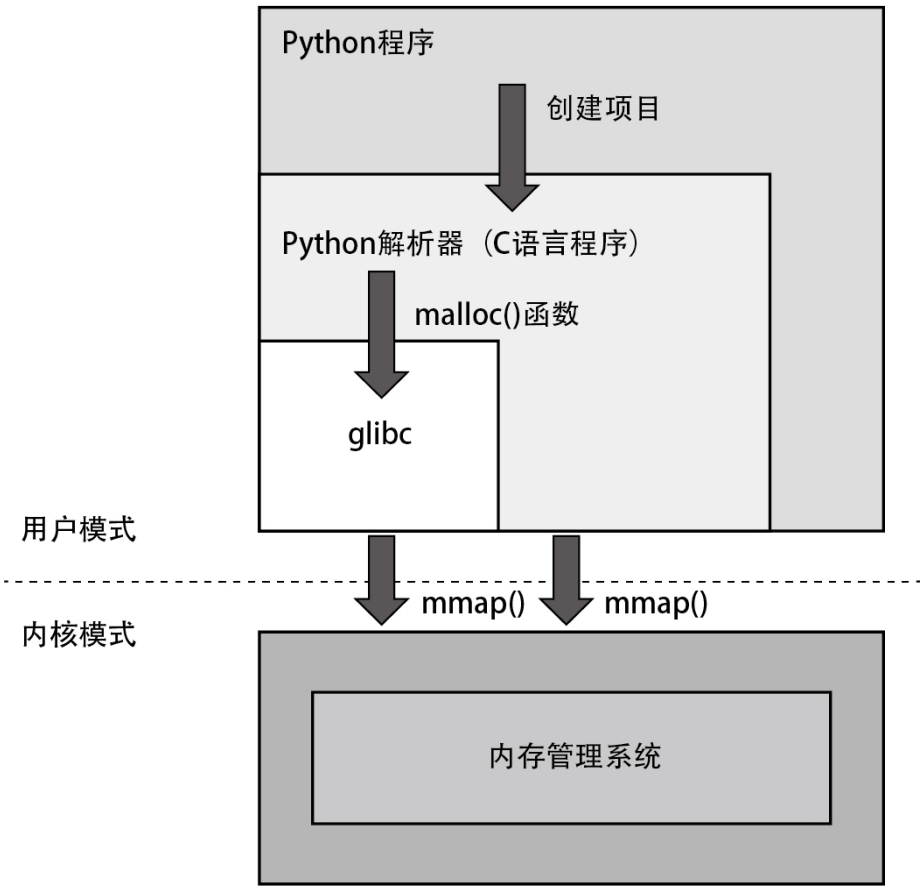

图 5-22　使用 Python 程序获取内存

对此感兴趣的读者可利用 `strace` 命令追踪 Python 脚本的运行。

5.10 解决问题

虽然我们对虚拟内存进行了大量说明，但是虚拟内存到底是怎样解决前面提到的问题的呢？

● 内存碎片化

如图 5-23 所示，假如能巧妙地设定进程的页表，就能将物理内存上的碎片整合成虚拟地址空间上的一片连续的内存区域。这样一来，碎片化的问题也就解决了。

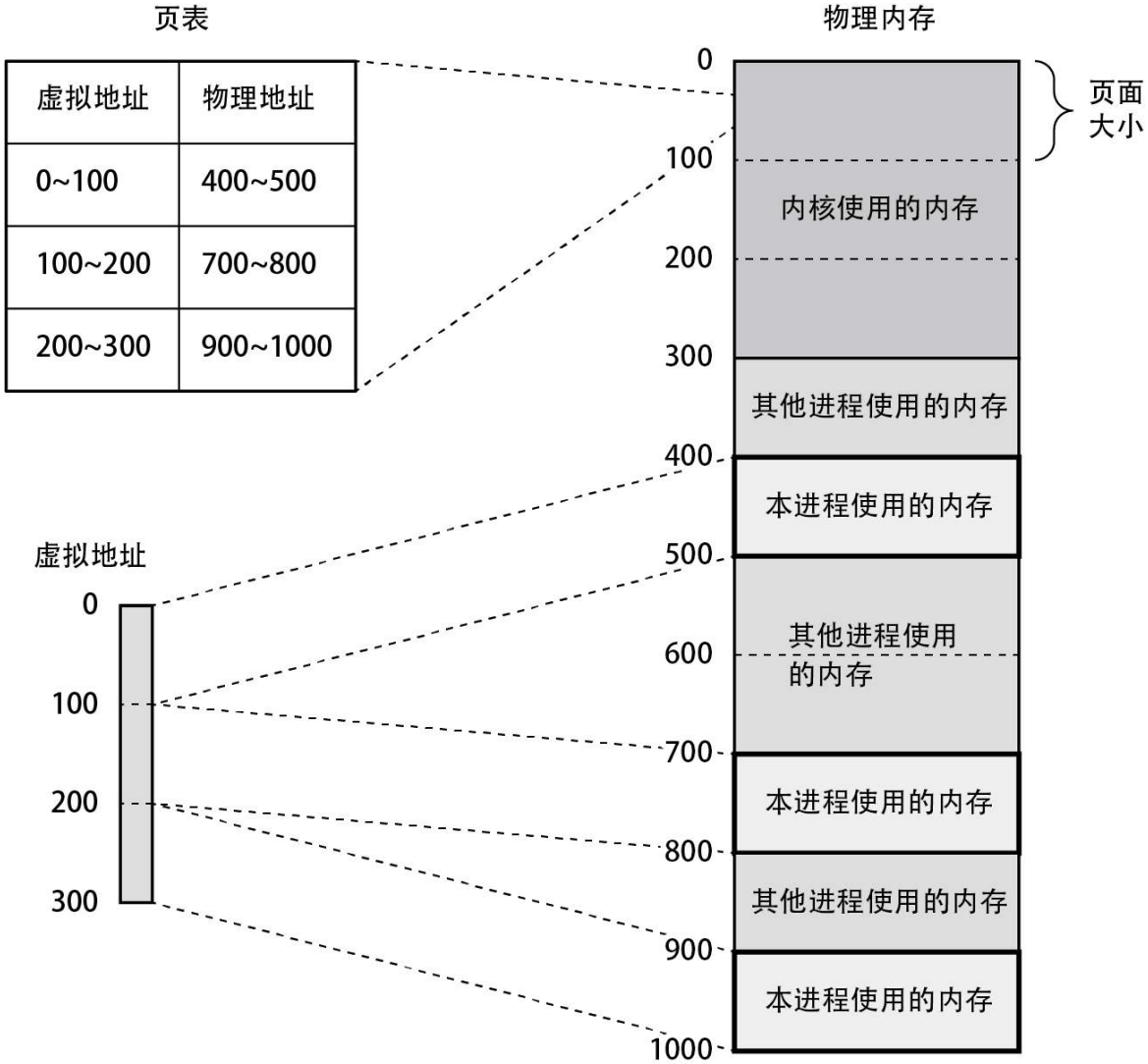

图 5-23　通过虚拟内存解决内存碎片化问题

● 访问用于其他用途的内存区域

虚拟地址空间是每个进程独有的。相应地，页表也是每个进程独有的。如图 5-24 所示，进程 A 和进程 B 各自拥有独立的虚拟地址空间。

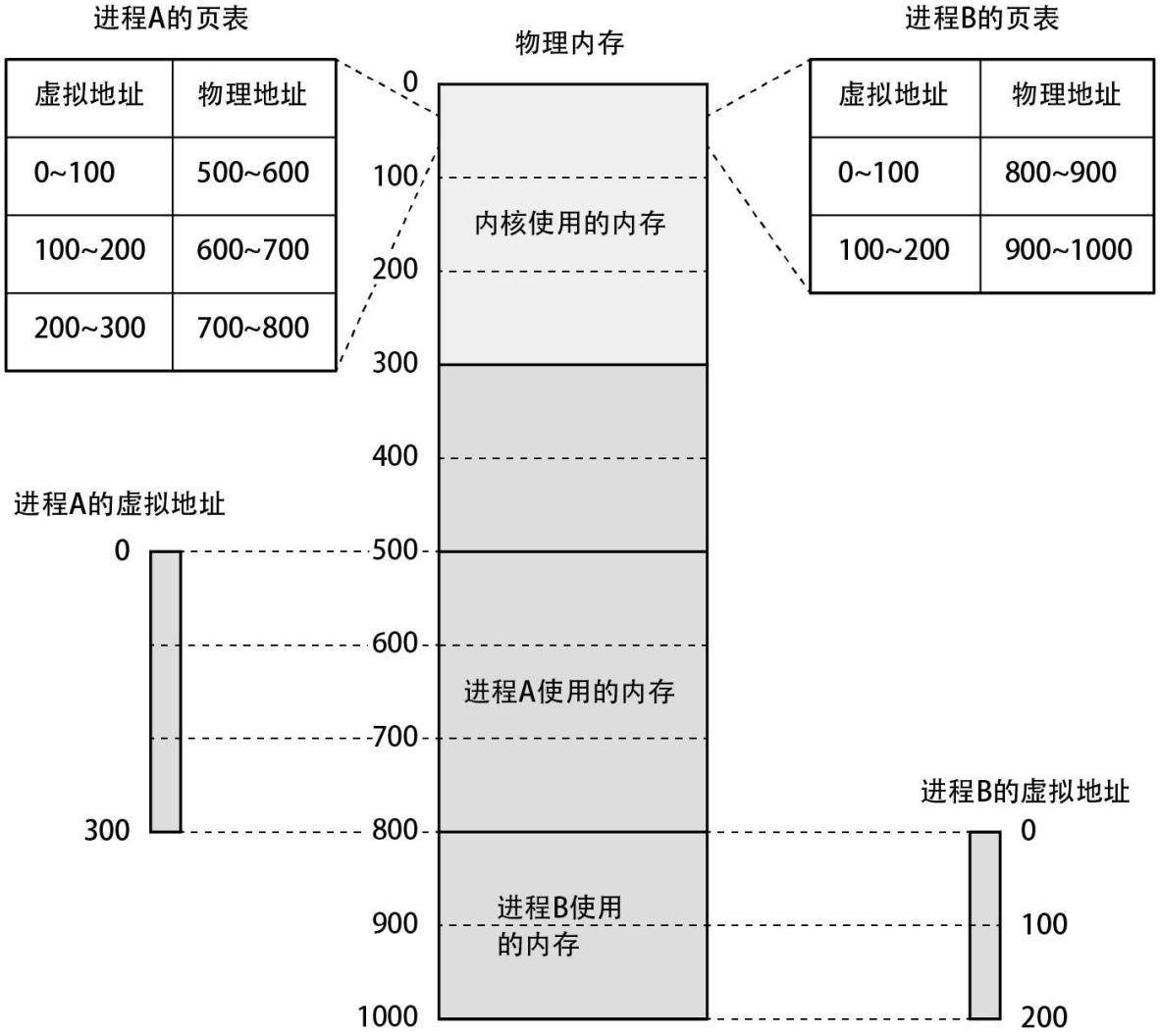

图 5-24　每个进程拥有独立的虚拟地址空间

得益于虚拟内存，进程根本无法访问其他进程的内存（图 5-25）。

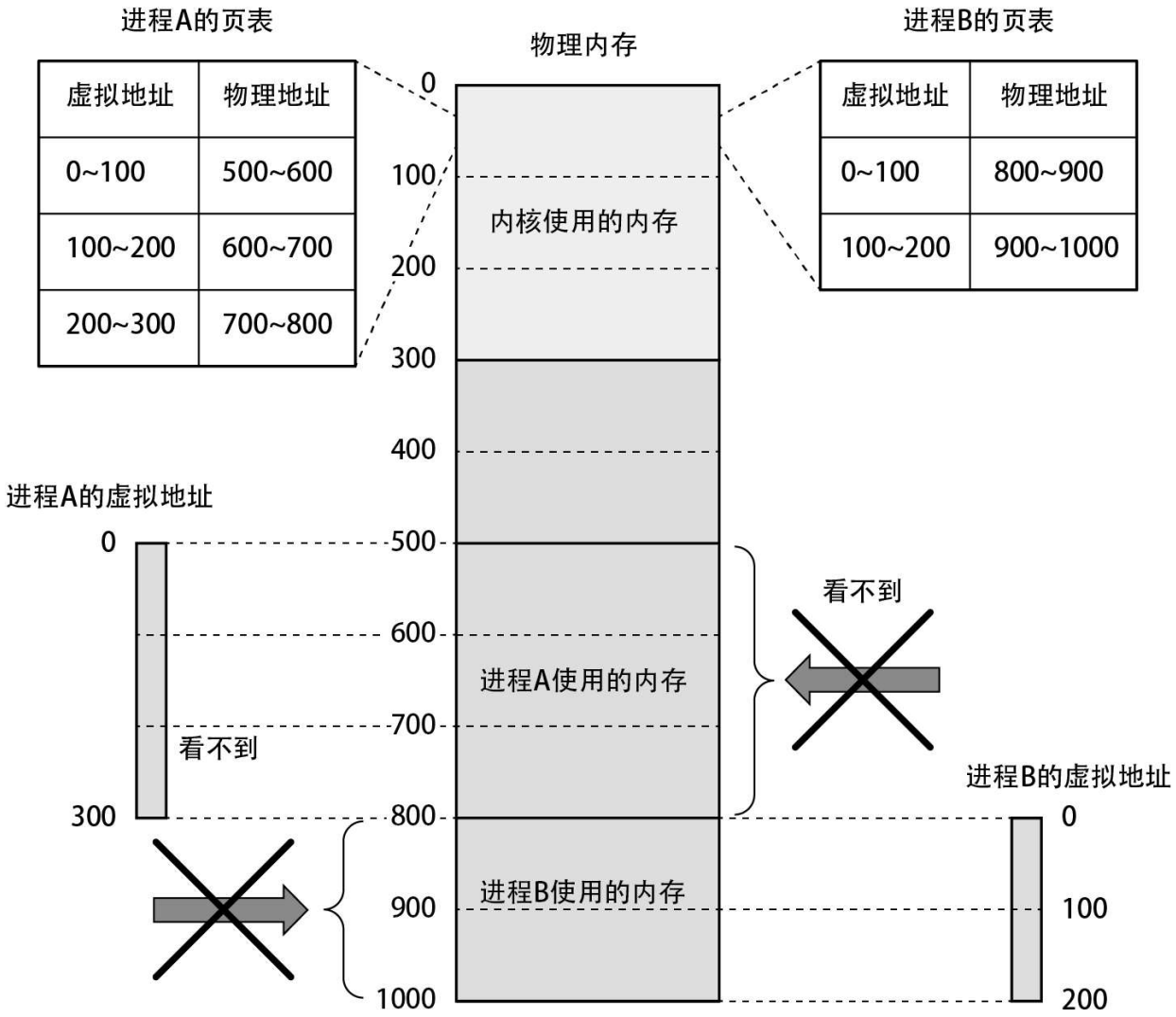

图 5-25　虚拟内存不允许进程访问其他进程的内存空间

出于实现上的方便，内核的内存区域被映射到了所有进程的虚拟地址空间中。但是，与内核的内存对应的页表项上都注有“内核模式专用”的信息，表明仅允许在 CPU 运行在内核模式下时访问，因此这部分内存也不可能被运行在用户模式下的进程窥探或损毁（图 5-26）。

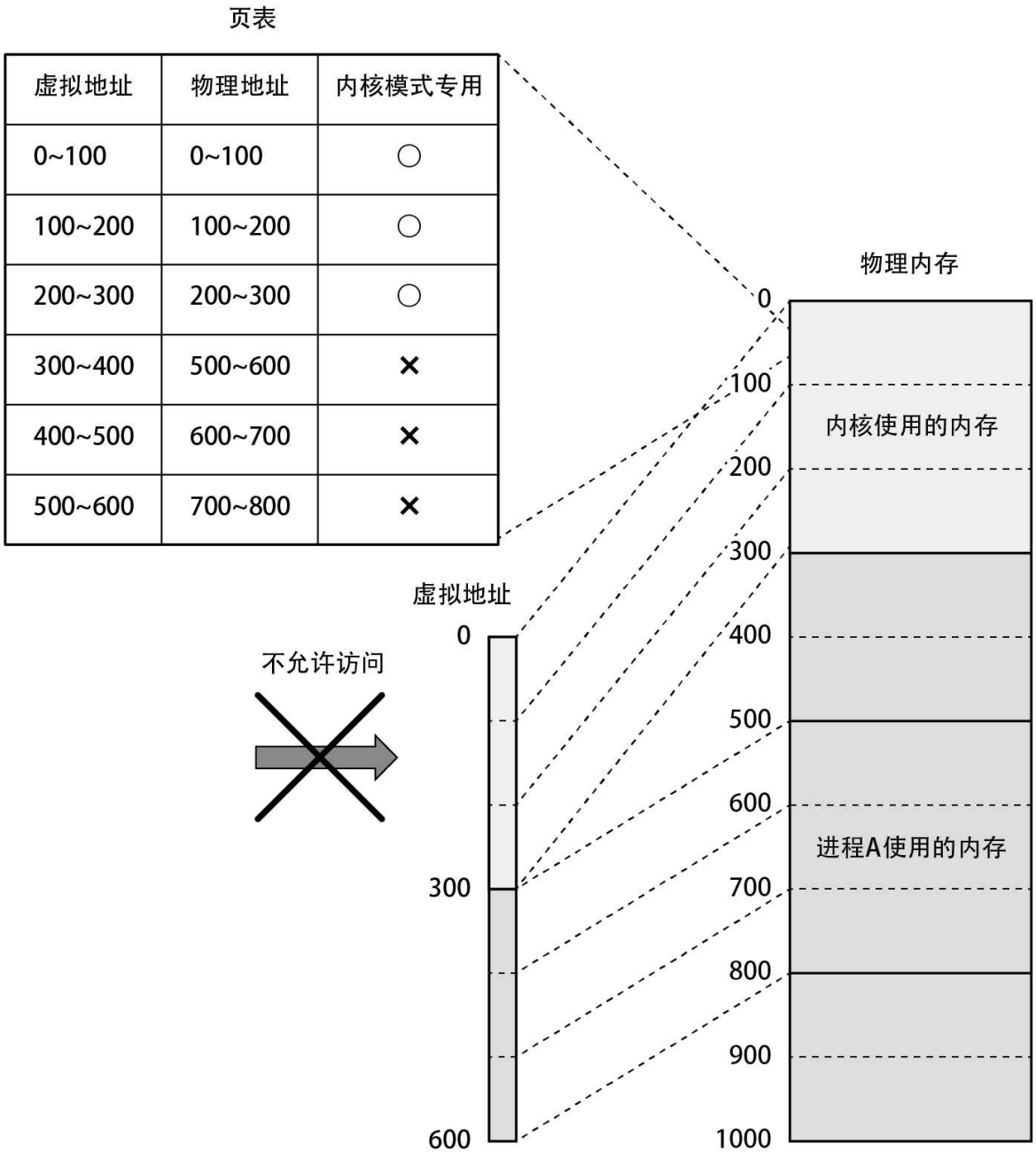

图 5-26　只有内核可以访问内核使用的内存

关于将内核的内存映射到进程的虚拟地址空间中的原因，因为不属于本书的范畴，所以这里不再过多说明。另外，此后的所有图表都将省略掉内核的内存区域的映射。

● 难以执行多任务

如前所述，每个进程拥有独立的虚拟地址空间。因此，我们可以编写运行于专用地址空间的程序，而不用担心干扰其他程序的运行，同时也不用关心自身的内存在哪个物理地址上（图 5-27）。

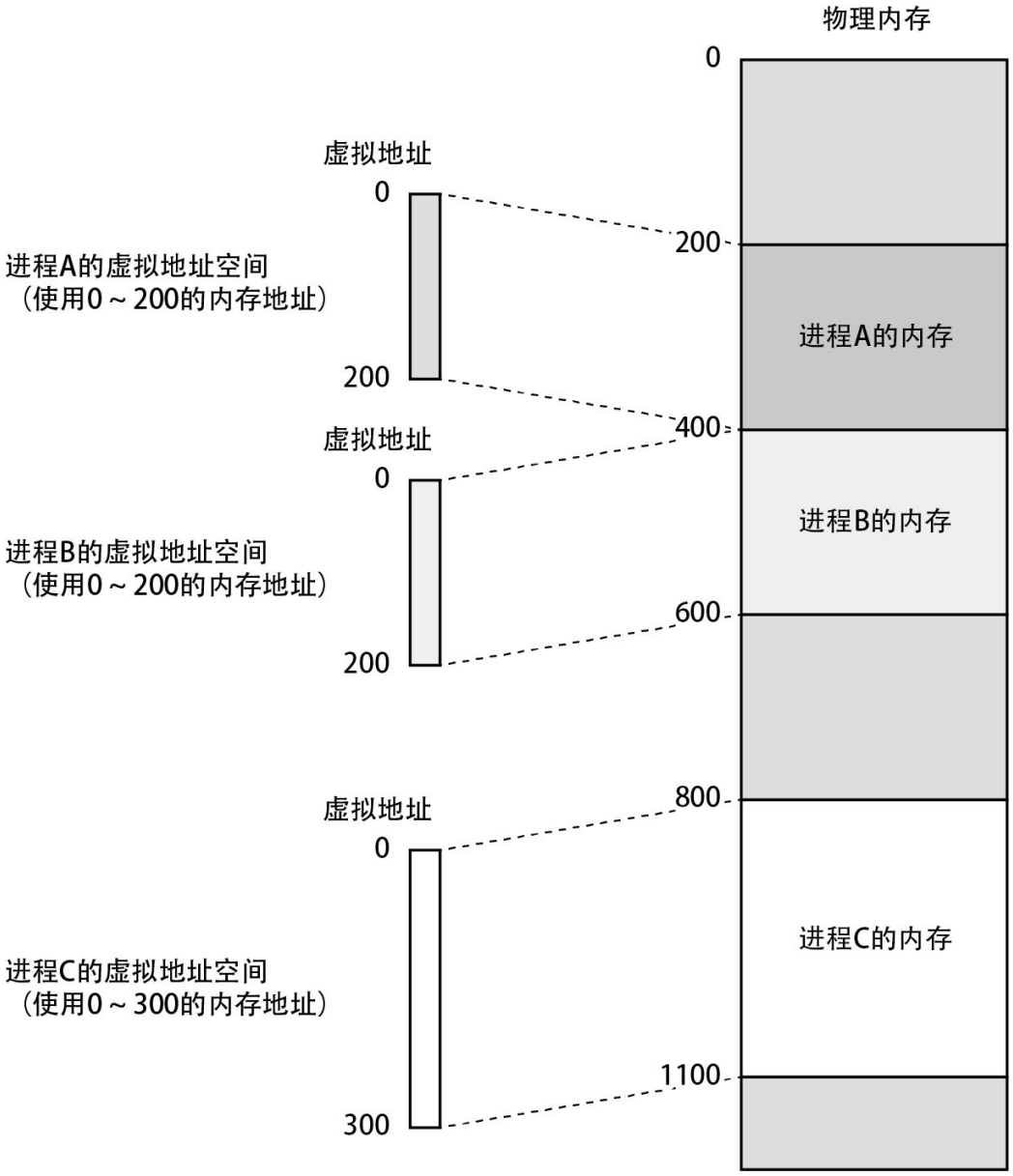

图 5-27　各个进程都无须关心自己被映射在哪个物理地址上

5.11 虚拟内存的应用

至此，我们已经介绍完虚拟内存的基本机制了，下面我们来介绍几个利用了虚拟内存机制的重要功能。

- 文件映射
- 请求分页
- 利用写时复制快速创建进程
- Swap
- 多级页表
- 标准大页

5.12　文件映射

进程在访问文件时，通常会在打开文件后使用 read()、write() 以及 lseek() 等系统调用。此外，Linux 还提供了将文件区域映射到虚拟地址空间的功能。

按照指定方式调用 mmap() 函数，即可将文件的内容读取到内存中，然后把这个内存区域映射到虚拟地址空间，如图 5-28 所示。

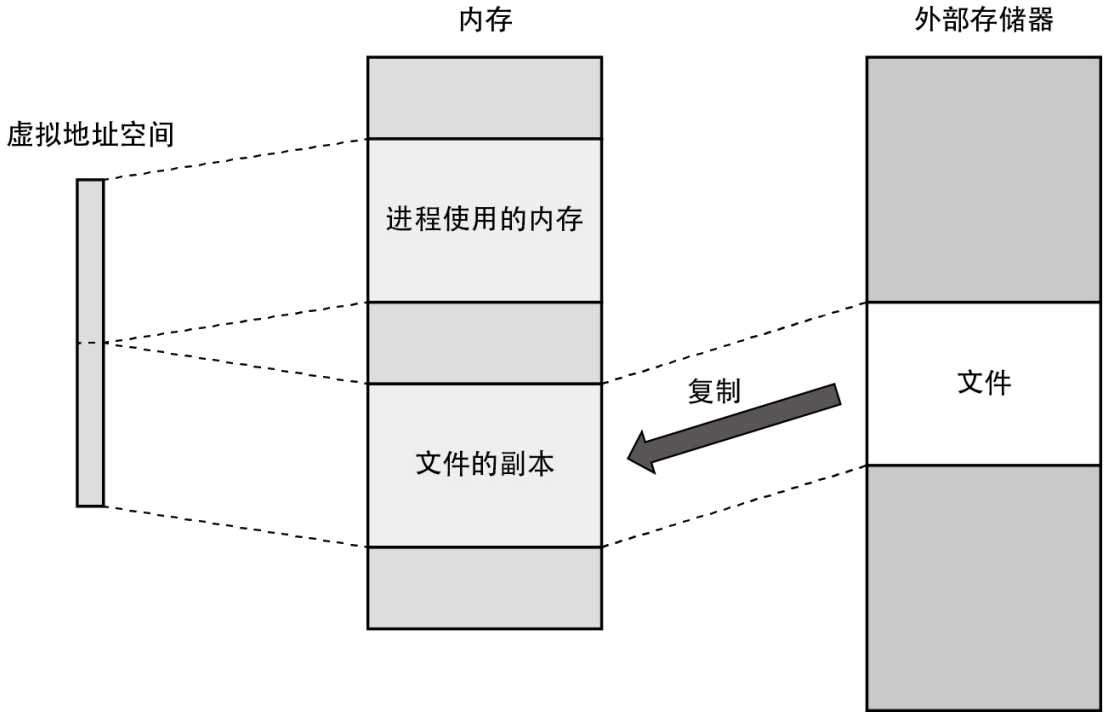

图 5-28　文件映射

这样就可以按照访问内存的方式来访问被映射的文件了。被访问的区域会在规定的时间点写入外部存储器上的文件（图 5-29）。关于这个时间点的内容，我们将在第 6 章说明。

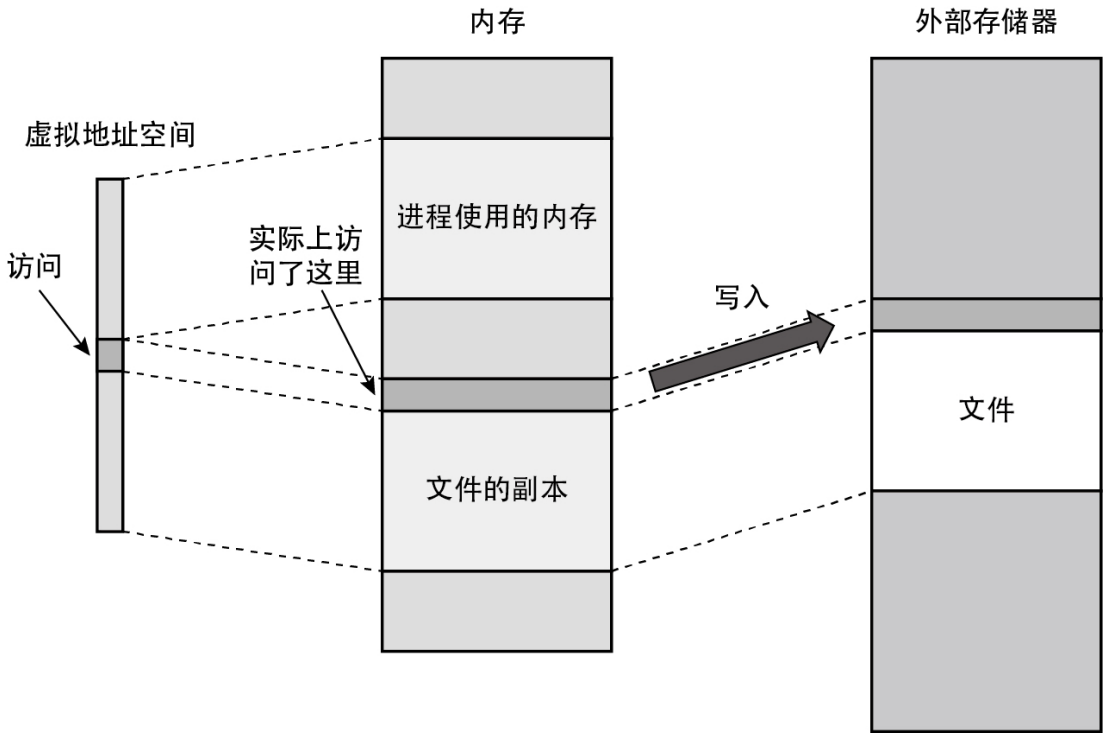

图 5-29　将访问过的区域写入文件中

● 文件映射的实验

现在编写一个使用文件映射功能的程序，来确认是否真的能映射文件，以及能否成功访问文件内容。需要确认的事项如下所示。

- 文件是否被映射到虚拟地址空间？
- 能否通过读取映射的区域来读取文件内容？
- 能否通过向映射的区域写入数据来将数据写入文件？

首先，创建一个名为 testfile 的文件，并向其写入字符串 hello。

```
$ echo hello >testfile
$
```

然后，编写实现下述要求的程序。

① 显示进程的内存映射信息（***proc***pid**/maps** 的输出）。

② 打开 testfile 文件。

③ 通过 **mmap()** 把文件映射到内存空间。

④ 再次显示进程的内存映射信息。

⑤ 读取并输出映射的区域中的数据。

⑥ 改写映射的区域中的数据。

完成后的程序如代码清单 5-3 所示。

代码清单 5-3 filemap 程序（filemap.c）

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <err.h>

#define BUFFER_SIZE    1000
#define ALLOC_SIZE     (100*1024*1024)

static char command[BUFFER_SIZE];
static char file_contents[BUFFER_SIZE];
static char overwrite_data[] = "HELLO";

int main(void)
{
    pid_t pid;

    pid = getpid();
    snprintf(command, BUFFER_SIZE, "cat proc%d/maps", pid);

    puts("*** memory map before mapping file ***");
    fflush(stdout);
    system(command);

    int fd;
    fd = open("testfile", O_RDWR);
    if (fd == -1)
        err(EXIT_FAILURE, "open() failed");

    char  file_contents;
    file_contents = mmap(NULL, ALLOC_SIZE, PROT_READ | PROT_
```

```
                        WRITE, MAP_SHARED, fd, 0);
    if (file_contents == (void ) -1) {
        warn("mmap() failed");
        goto close_file;
    }

    puts("");
    printf("*** succeeded to map file: address = %p; size =
           0x%x ***\n", file_contents, ALLOC_SIZE);

    puts("");
    puts("*** memory map after mapping file ***");
    fflush(stdout);
    system(command);

    puts("");
    printf("*** file contents before overwrite mapped region:
           %s", file_contents);

    // 覆写映射的区域
    memcpy(file_contents, overwrite_data, strlen(overwrite_
           data));

    puts("");
    printf("*** overwritten mapped region with: %s\n", file_
           contents);

    if (munmap(file_contents, ALLOC_SIZE) == -1)
        warn("munmap() failed");
close_file:
    if (close(fd) == -1)
        warn("close() failed");
    exit(EXIT_SUCCESS);
}
```

编译并运行这个程序，结果如下。

```
$ cc -o filemap filemap.c
$ ./filemap
*** memory map before mapping file ***
( 略 )
*** succeeded to map file: address = 0x7fc8cd24d000;       ↲
                              size = 0x6400000 ***      ←①
*** memory map after mapping file ***
( 略 )
7fc8cd24d000-7fc8d364d000  rw-s  00000000  00:16  142745640
homesat/work/book/st-book-kernel-in-practice 05-memory- ↲
managementsrc/testfile                                  ←②
( 略 )
```