即使如图 5-10 所示，将程序强行映射到其他地方（在该图中为内存地址 800 与 1100 的位置）来运行，也会因为代码和数据指向的内存地址与预期不同而无法正常运行。不仅如此，内存中属于其他进程或内核的区域也可能遭到损毁。

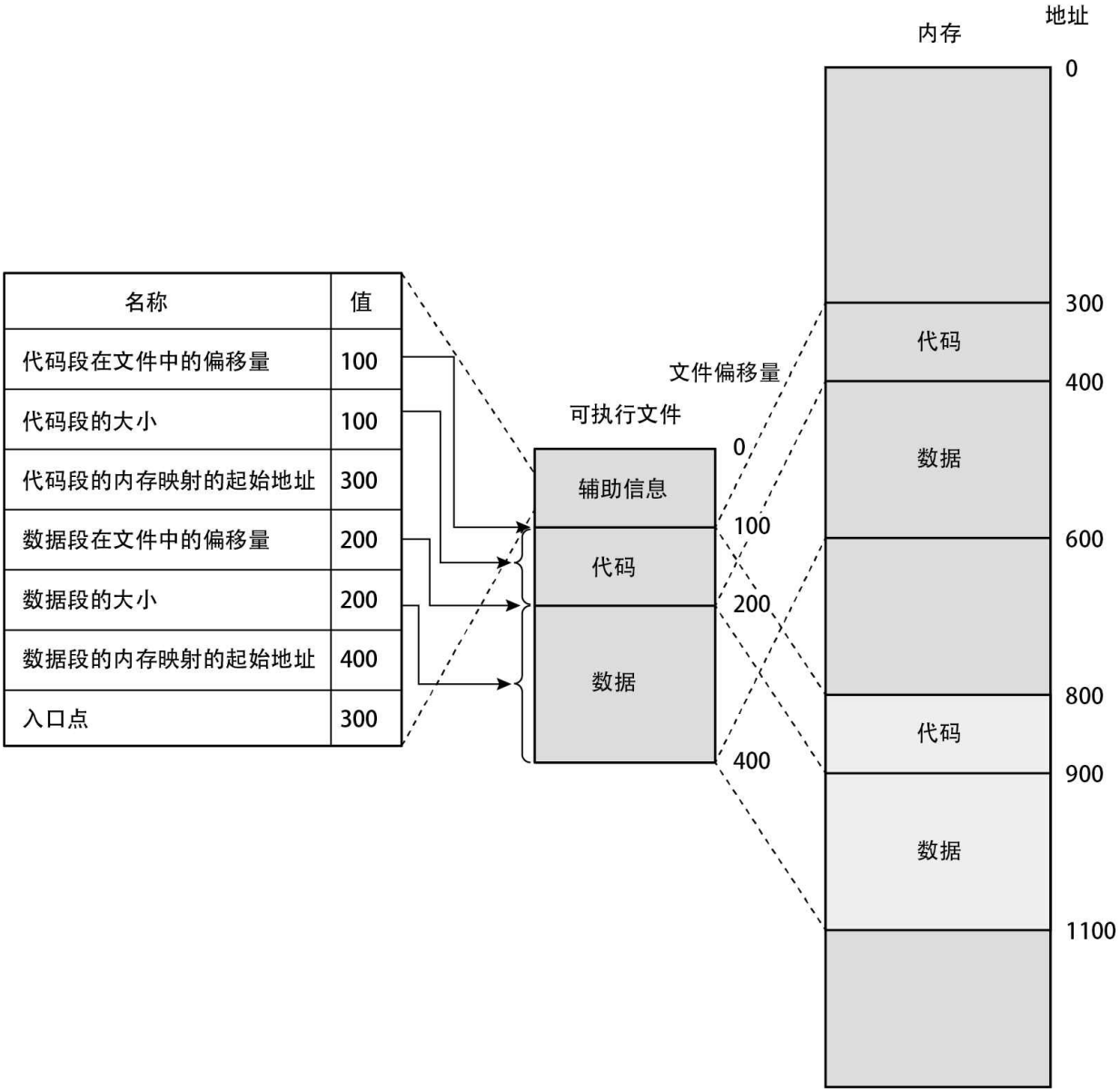


图 5-10　再次启动同一个程序时的情景

这也适用于启动两个不同的程序的情况。在使用这种简单的内存分配方式时，我们在编写程序时，就不得不注意防止与其他程序的内存地址出现重合。

◆ ◆ ◆

可以看到，简单的内存分配方式存在各种尚待解决的问题。下面我们将解释如何通过引入虚拟内存机制，把这些问题一口气解决掉。

# 5.4　虚拟内存

为了解决上一节罗列的问题，现代 CPU 搭载了被称为**虚拟内存**的功能，Linux 也利用了这一功能。

简而言之，虚拟内存使进程无法直接访问系统上搭载的内存，取而代之的是通过虚拟地址间接访问。进程可以看见的是**虚拟地址**，系统上搭载的内存的实际地址称为**物理地址**。此外，可以通过地址访问的范围称为**地址空间**（图 5-11）。

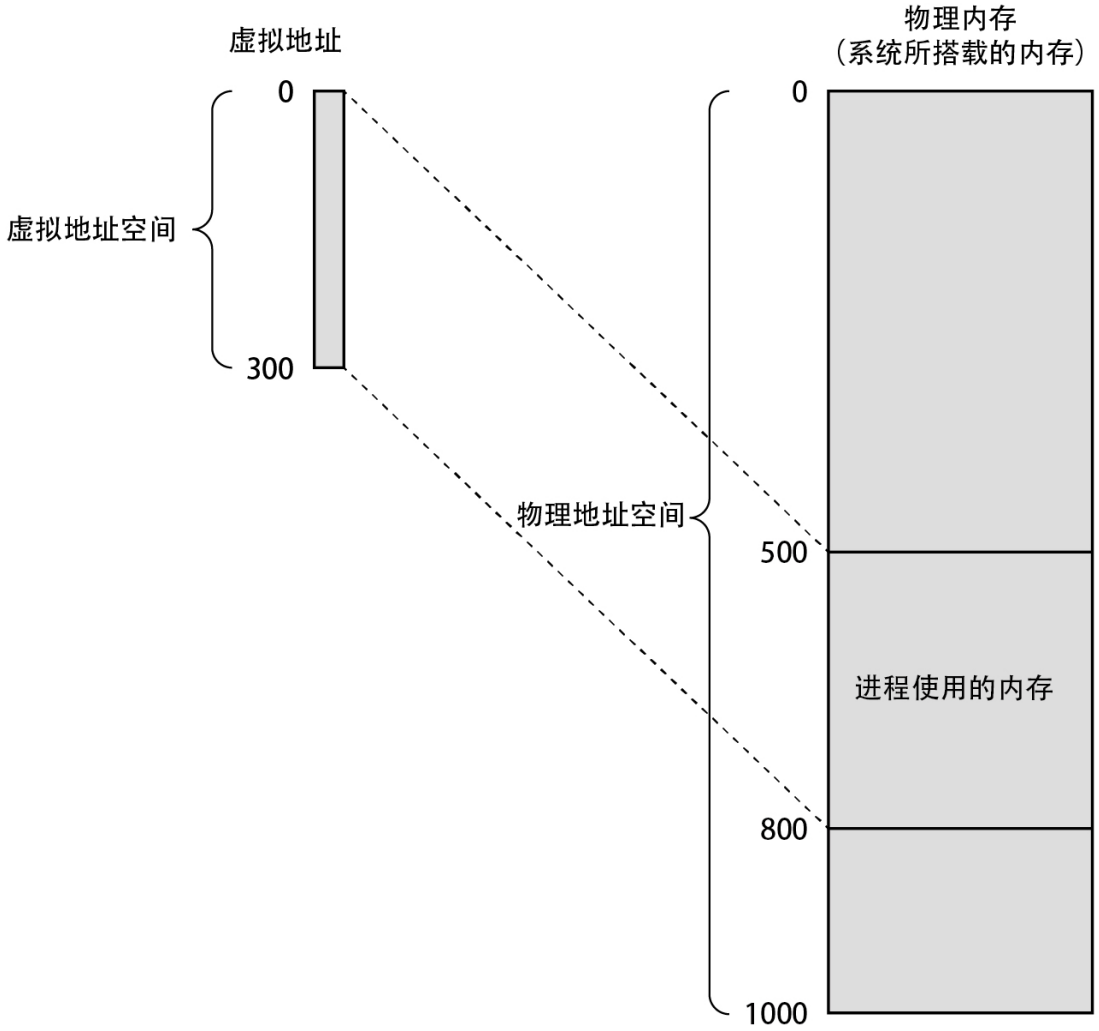


图 5-11　虚拟内存

突然出现的专有名词可能会导致这部分内容难以理解，但要想理解后面的内容，这部分知识是不可或缺的。如果在后面的阅读中感到困惑，请回来重新看看图 5-11，整理一下思路。

假如进程在图 5-11 的状态下访问地址 100 的内存，则它实际上访问的是物理地址 600 的内存上的数据，如图 5-12 所示。

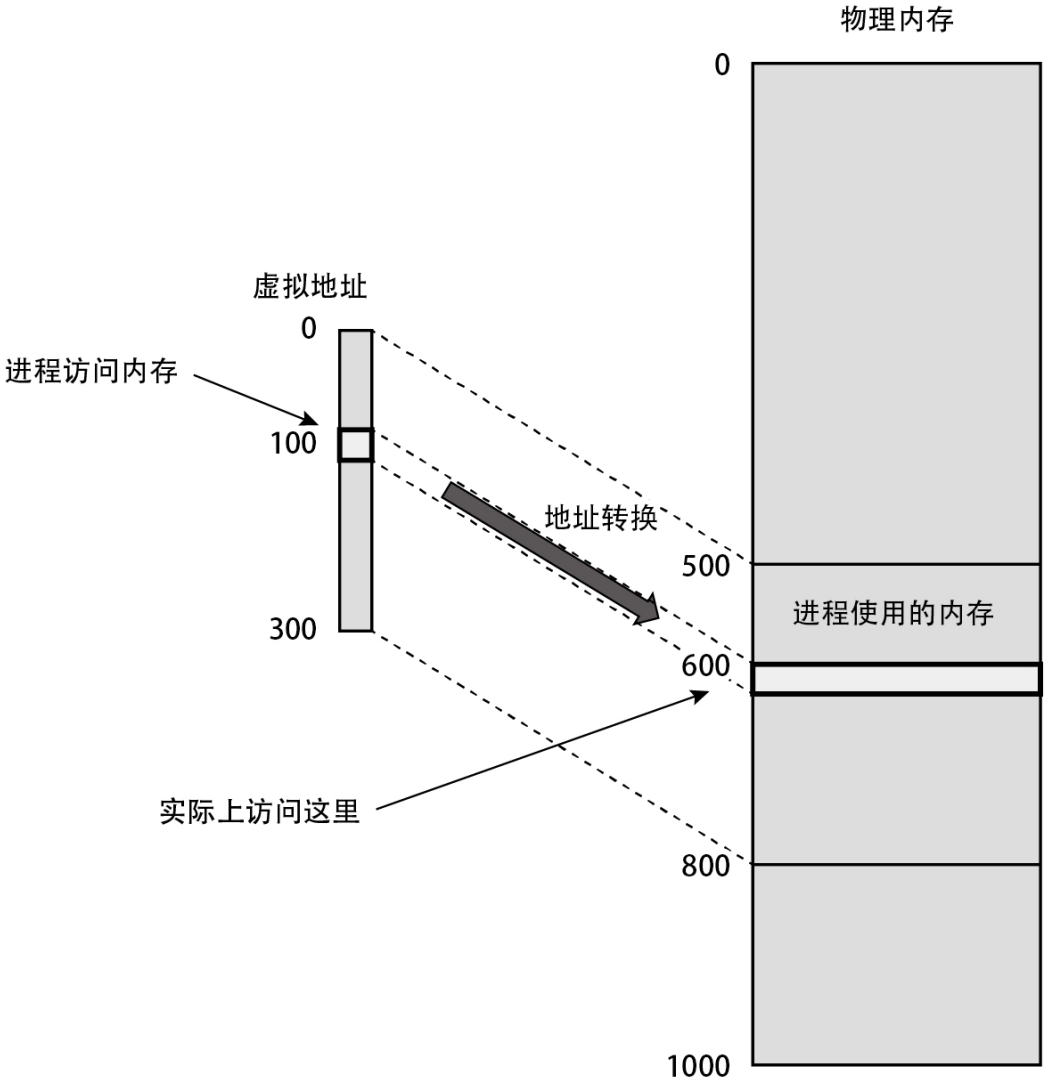


图 5-12　访问虚拟地址 100

我们在第 3 章中通过 `readelf` 命令或者 `cat` *proc*pid/maps 输出的地址也是虚拟地址。另外，进程无法直接访问真实的内存，也就是说不存在直接访问物理地址的方法。

# 5.5 页表

通过保存在内核使用的内存中的**页表**，可以完成从虚拟地址到物理地址的转换。在虚拟内存中，所有内存以页为单位划分并进行管理，地址转换也以页为单位进行。在页表中，一个页面对应的数据条目称为**页表项**。页表项记录着虚拟地址与物理地址的对应关系。

页面大小取决于 CPU 架构。在 x86_64 架构中，页面大小为 4 KB。但为了便于说明，本书假设一个页面的大小为 100 字节。

虚拟地址 0 *300 映射到物理地址 500* 800 的情形如图 5-13 所示。

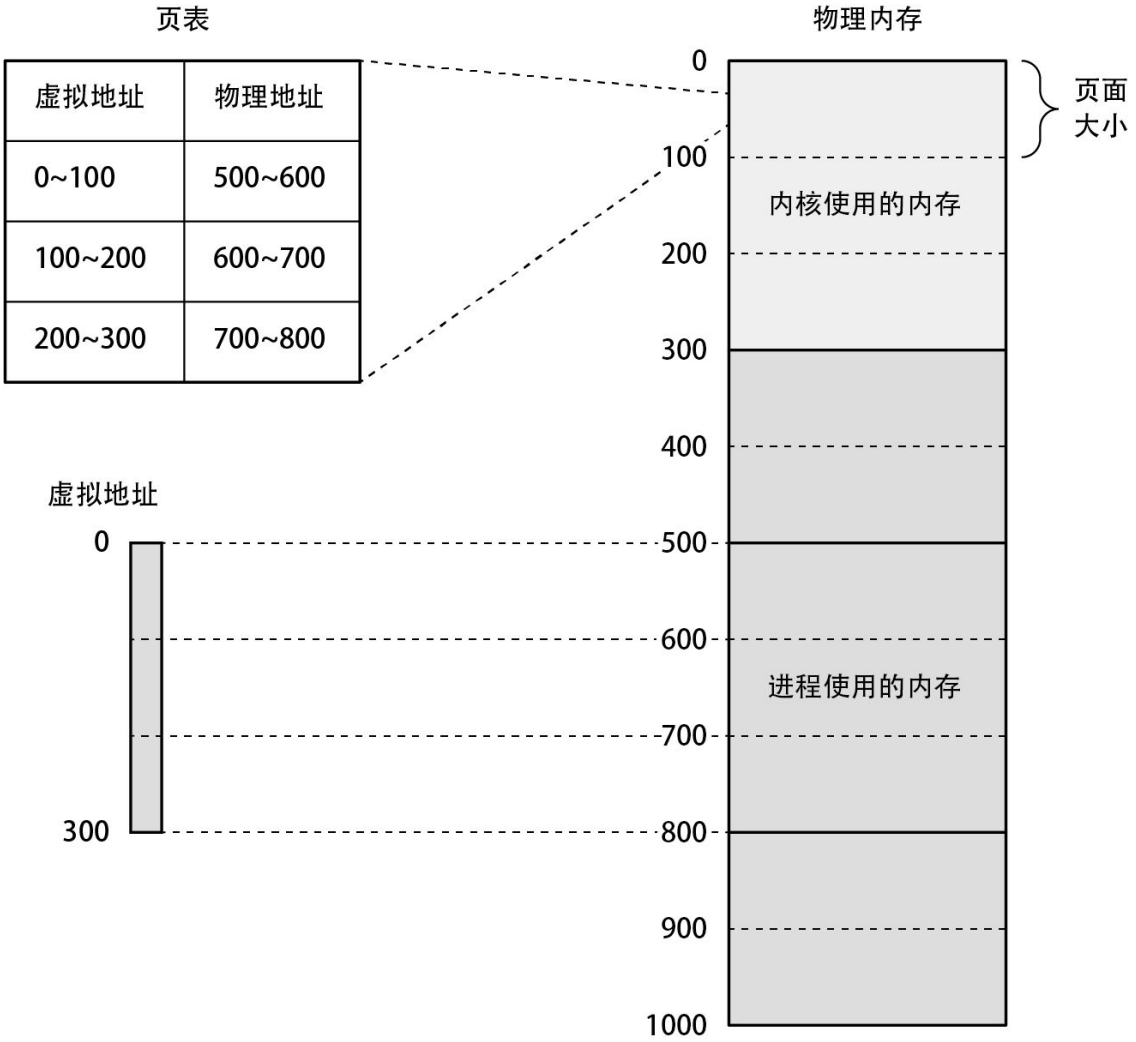

图 5-13　通过页表将虚拟地址映射到物理地址

如果进程访问 0 ~ 300 的虚拟地址，CPU 将自动参考页表的内容，将其转换为对相应的物理地址的访问，而无须经过内核的处理。

访问 300 以后的虚拟地址，会发生什么呢？实际上，虚拟地址空间的大小是固定的，并且页表项中存在一个表示页面是否关联着物理内存的数据。虚拟地址空间的大小为 500 字节时的情形如图 5-14 所示。

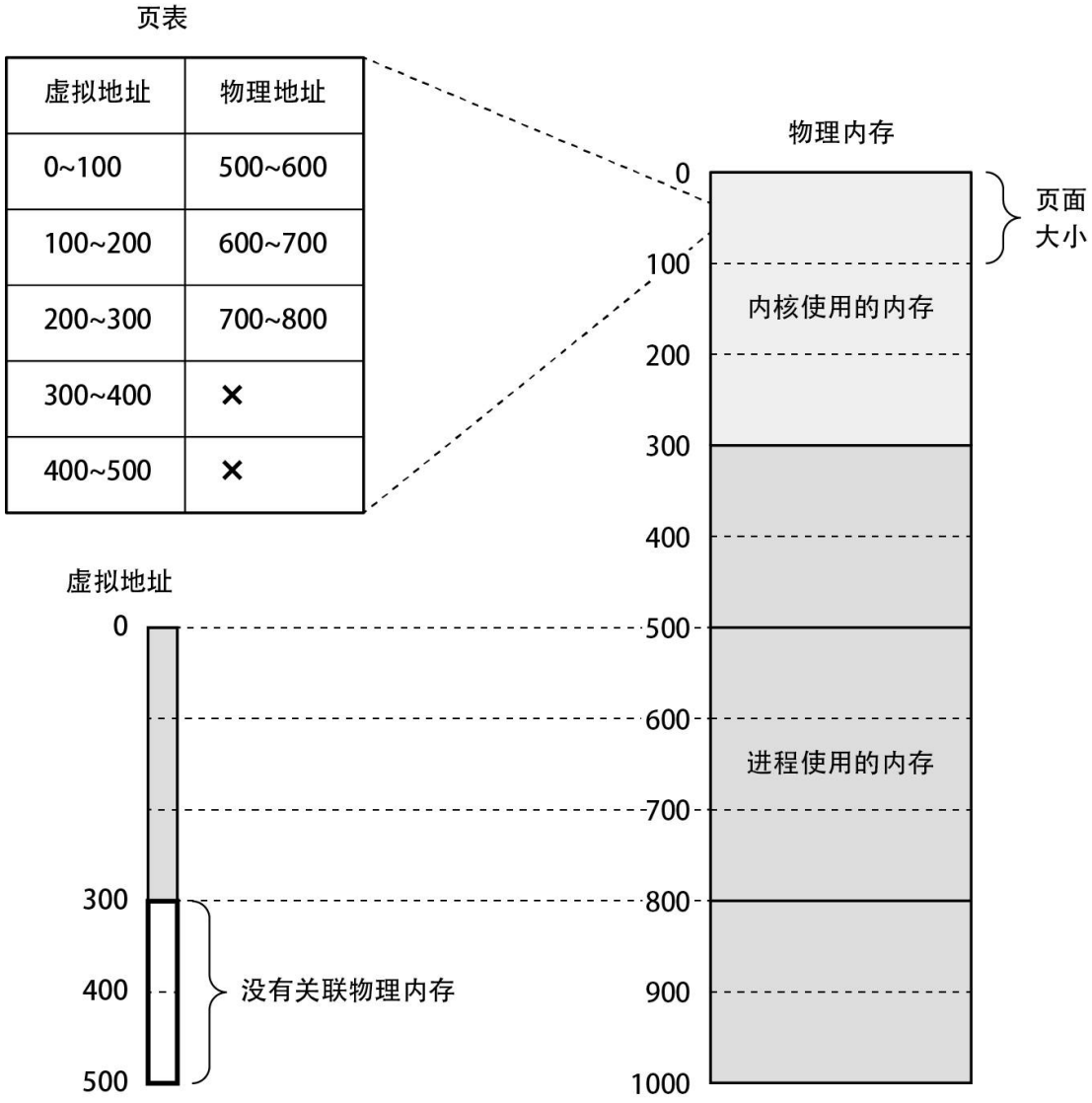


图 5-14　页表（没有为地址 300 ~ 500 分配物理内存）

如果进程访问地址 300 ~ 500，则在 CPU 上会发生**缺页中断**。缺页中断可以中止正在执行的命令，并启动内核中的**缺页中断机构**的处理。例如，在图 5-14 的状态下访问地址 300 时的情形如图 5-15 所示。

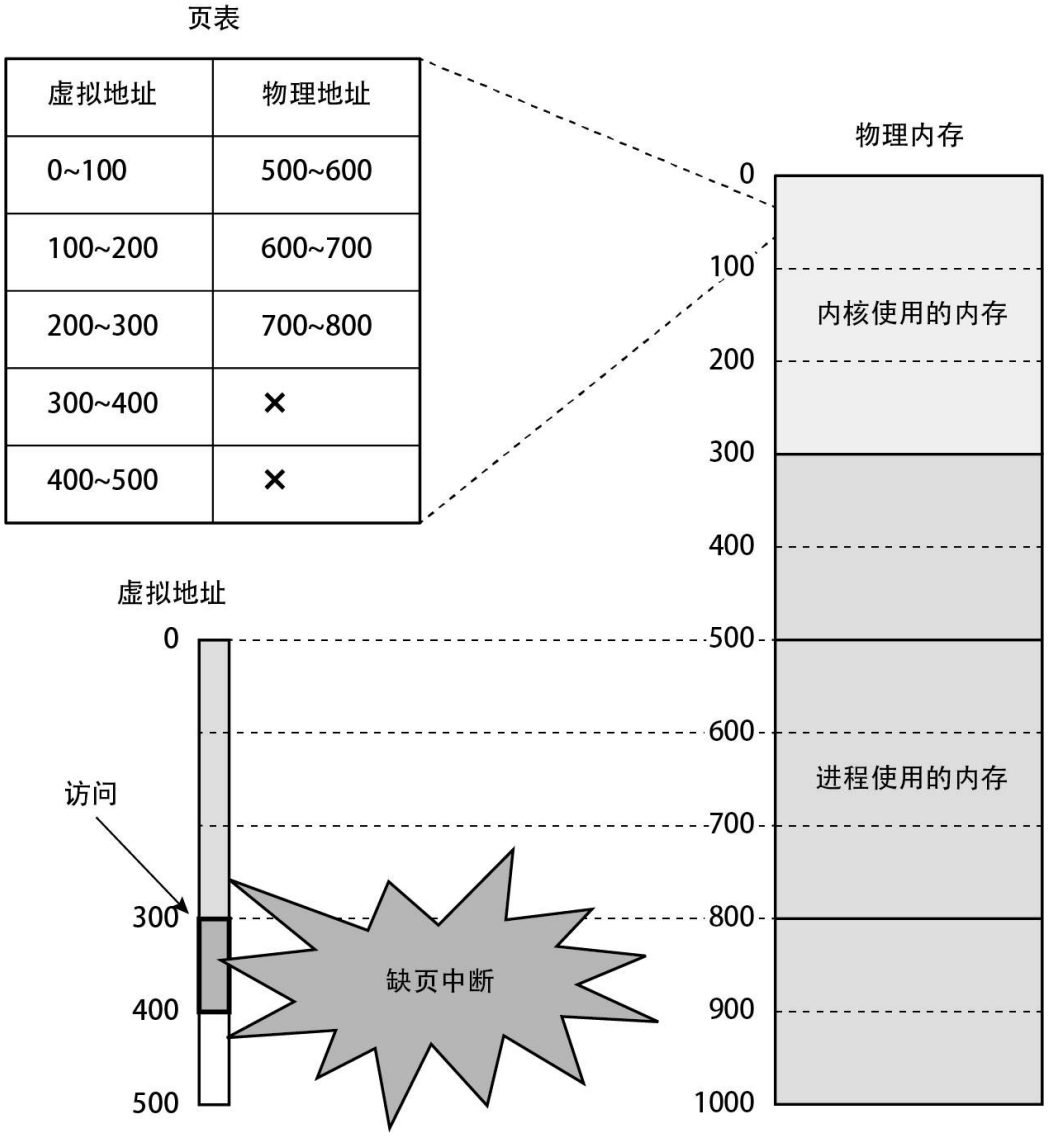


图 5-15　引发缺页中断

内核的缺页中断机构检测到非法访问，向进程发送 SIGSEGV 信号。接收到该信号的进程通常会被强制结束运行。

## 5.6 实验

下面来编写一个访问非法地址的程序，程序的要求如下所示。

① 输出字符串 **before invalid access**。

② 向必定会访问失败的地址 **NULL** 写入一个值（这里将写入 **0**）。

③ 输出字符串 **after invalid access**。

满足上述要求的程序的实现如代码清单 5-1 所示。

代码清单 5-1 segv 程序（segv.c）

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p = NULL;
    puts("before invalid access");
    *p = 0;
    puts("after invalid access");
    exit(EXIT_SUCCESS);
}
```

运行这个程序，结果如下。

```
$ cc -o segv segv.c
$ ./segv
before invalid access
Segmentation fault (core dumped)
$
```

在输出字符串 before invalid access 后，程序并没有输出 after invalid access，而是输出了“Segmentation fault...”这一信息，然后就结束运行了。程序在输出 before invalid access 后向非法地址发出了访问，进而引发了 SIGSEGV 信号。由于没有对这一信号进行处理，所以该程序没有继续执行后续的代码就异常终止了。相信很多人遇到过与此类似的异常终止。

在使用 C 语言等直接操作内存地址的编程语言编写的程序中，上述问题的原因通常在于程序自身或者程序所使用的库。与此相对，在使用 Python 等并不直接操作内存地址的编程语言编写的程序中，问题的原因通常在于解析器或者程序依赖的库。

# 5.7　为进程分配内存

内核是如何利用虚拟内存机制为进程分配内存的呢？下面，我们一起来看看在创建进程时以及在进程创建后动态分配内存时的情形。

## ● 在创建进程时

首先读取程序的可执行文件，以及第 3 章中说明过的辅助信息。假设可执行文件的结构如下表所示。

| 名称 | 值 |
|---|---|
| 代码段在文件中的偏移量 | 100 |
| 代码段的大小 | 100 |
| 代码段的内存映射的起始地址 | 0 |
| 数据段在文件中的偏移量 | 200 |
| 数据段的大小 | 200 |
| 数据段的内存映射的起始地址 | 100 |
| 入口点 | 0 |

运行程序所需的内存大小为：

$$\text{代码段的大小} + \text{数据段的大小} = 100 + 200 = 300$$

因此，在物理内存上划分出大小为 300 的区域，将其分配给进程，并把代码和数据复制过去（图 5-16）。

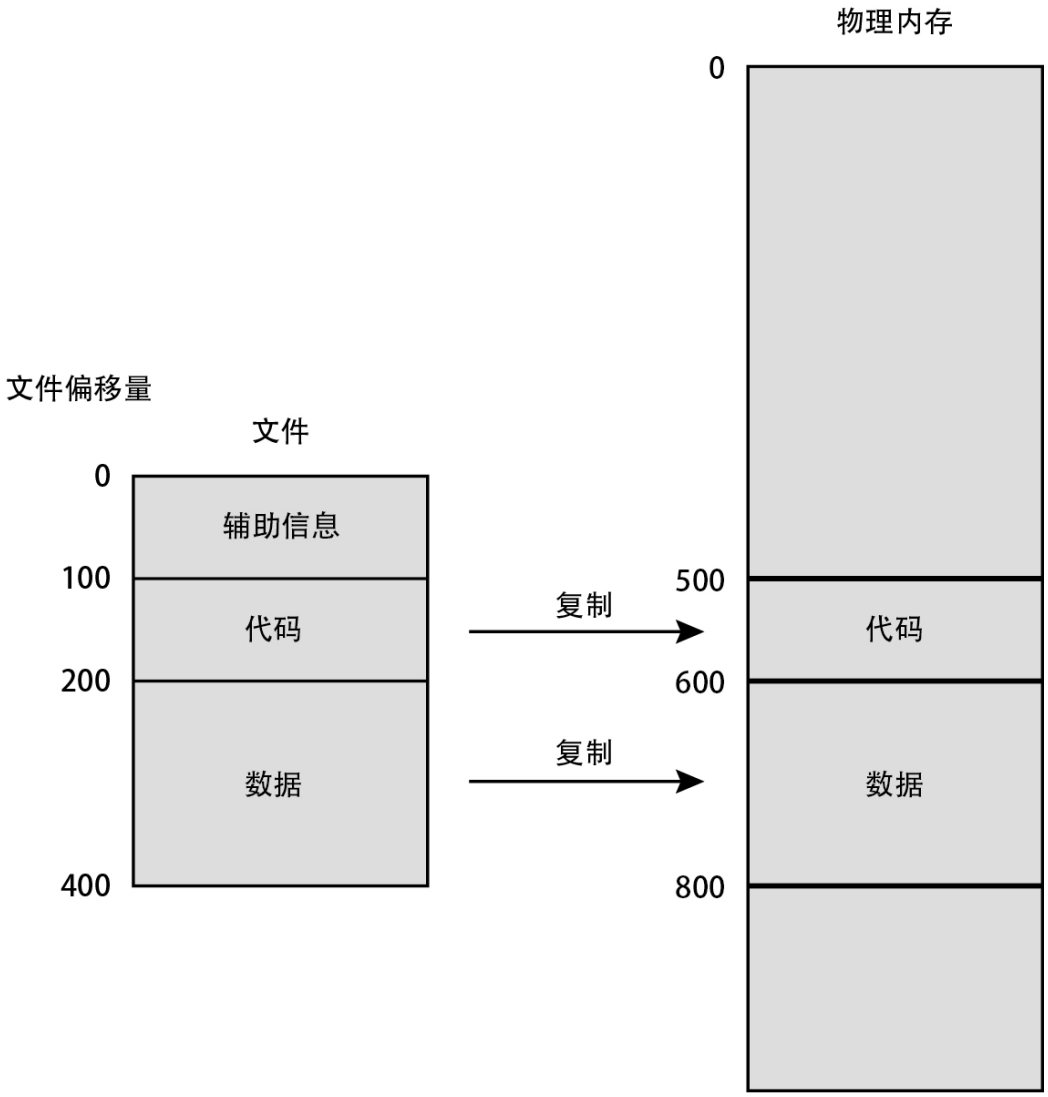


图 5-16　向进程分配内存（存在虚拟内存时的情形）

在现实中，Linux 的物理内存分配使用的是更复杂的**请求分页**方法。关于这部分内容，本章后文将进行说明。

在复制完成后，创建进程的页表，并把虚拟地址映射到物理地址（图 5-17）。

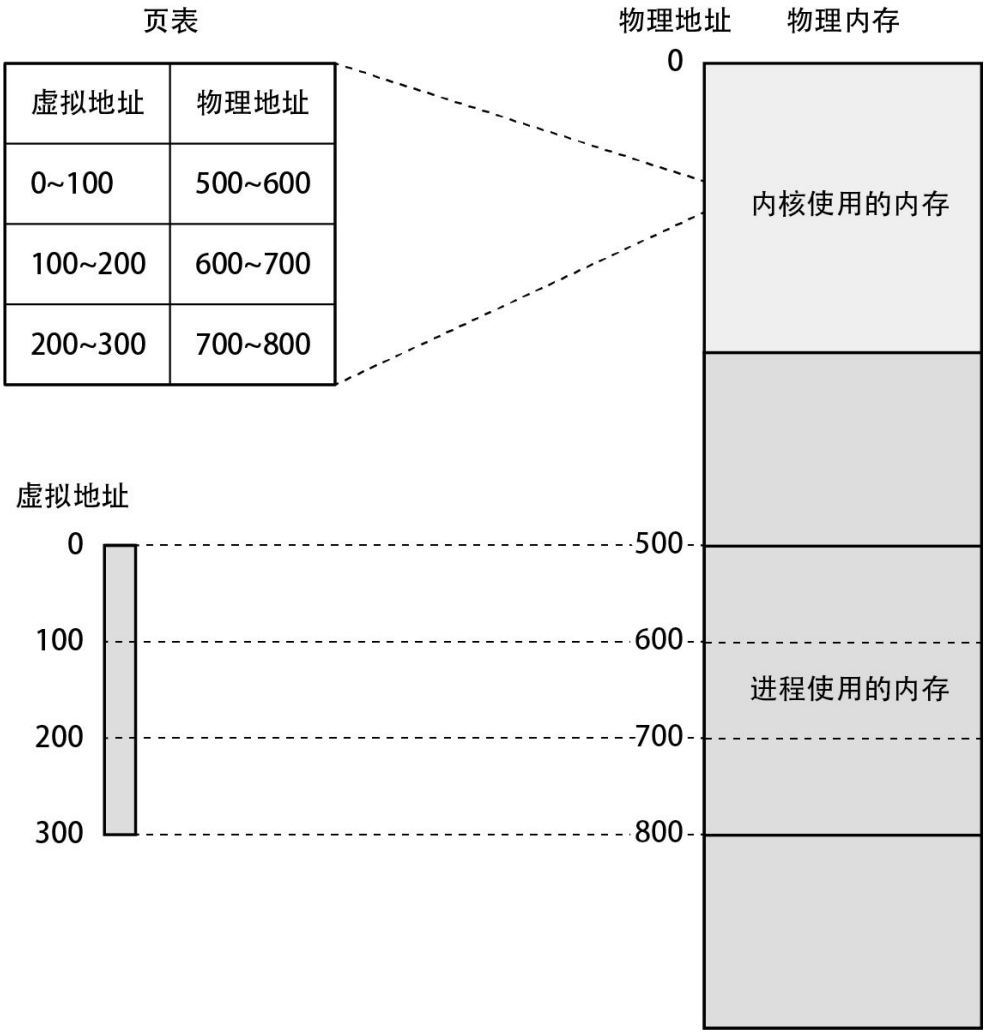


图 5-17　把虚拟地址映射到物理地址

最后，从指定的地址开始运行即可（图 5-18）。

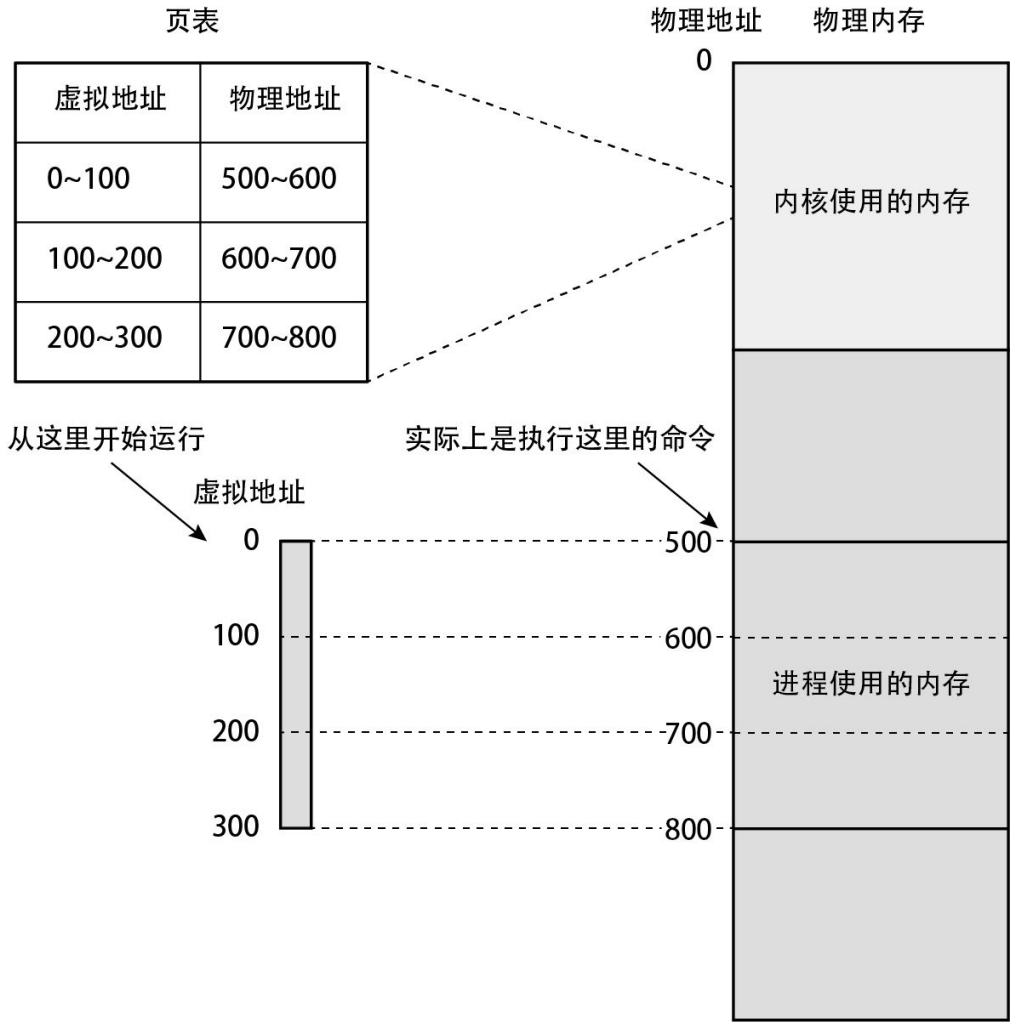


图 5-18　运行进程（存在虚拟内存的情形）

## ● 在动态分配内存时

如果进程请求更多内存，内核将为其分配新的内存，创建相应的页表，然后把与新分配的内存（的物理地址）对应的虚拟地址返回给进程。在图 5-17 的状态下请求新的 100 字节内存时的情形如图 5-19 所示。

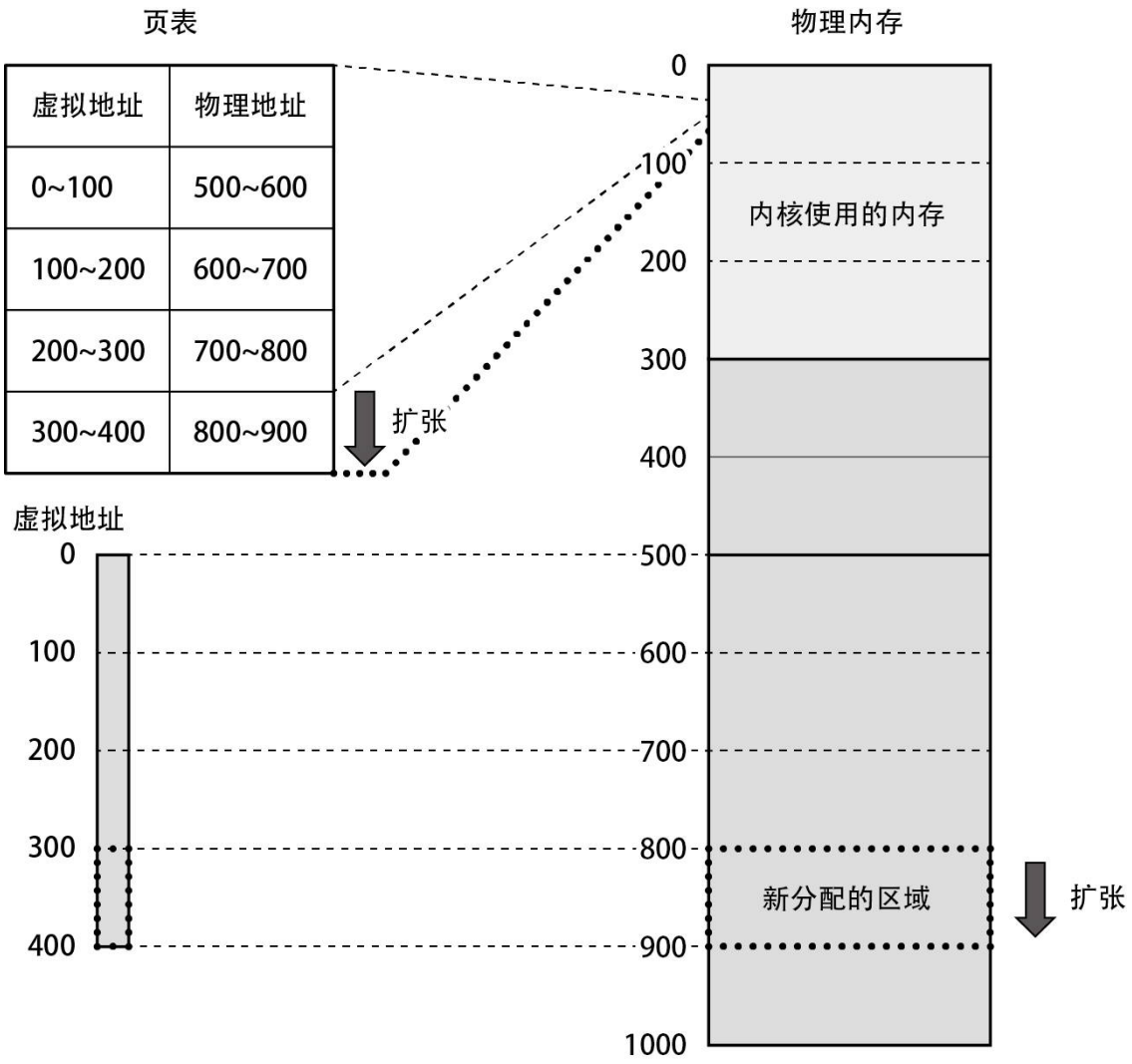


图 5-19　动态分配内存（存在虚拟内存时的情形）

## 5.8 实验

下面，我们来编写实现下述要求的程序，以确认内存分配的运作方式。

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

② 额外获取 100 MB 的内存。

③ 再次显示内存映射信息。

完成后的程序如代码清单 5-2 所示。

代码清单 5-2 mmap 程序（mmap.c）

```
#include <unistd.h>
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <err.h>

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

static char command[BUFFER_SIZE];

int main(void)
{
    pid_t pid;

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

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

    void new_memory;
    new_memory = mmap(NULL, ALLOC_SIZE, PROT_READ | PROT_WRITE,
                      MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (new_memory == (void ) -1)
        err(EXIT_FAILURE, "mmap() failed");

    puts("");
```