虚拟内存不足与剩余多少物理内存无关。如果不清楚虚拟内存的机制，可能难以想象这到底是一种什么样的情景。

在 x86 架构上，虚拟地址空间仅有 4 GB，因此数据库之类的大型程序经常会引发虚拟内存不足；但是在 x86_64 架构上，由于虚拟地址空间扩充到了 128 TB，所以虚拟内存不足变得非常罕见。但是，随着程序对虚拟内存的需求不断增加，我们可能会再次迎来容易引发虚拟内存不足的一天。

与虚拟内存不足相对的物理内存不足指的是系统上搭载的物理内存被耗尽的状态（图 5-36）。

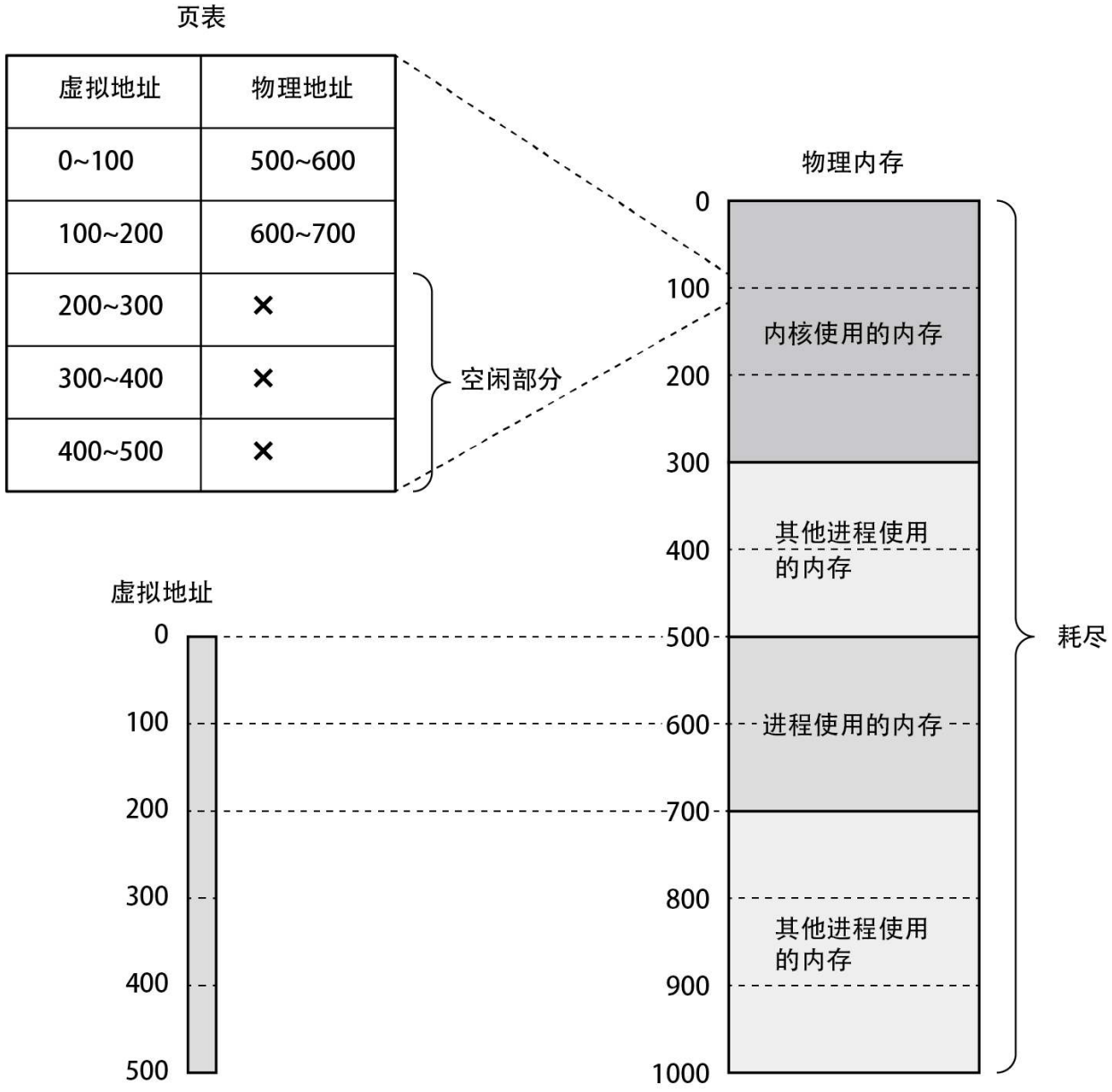

图 5-36　物理内存不足

物理内存不足与进程的虚拟内存剩余多少无关。与虚拟内存不足相比，物理内存不足的情形应该更容易想象。

5.14 写时复制

我们在第 3 章中介绍过用于创建进程的 fork() 系统调用，利用虚拟内存机制，可以提高 fork() 的执行速度。

在发起 fork() 系统调用时，并非把父进程的所有内存数据复制给子进程，而是仅复制父进程的页表。如图 5-37 所示，虽然在父进程和子进程双方的页表项内都存在表示写入权限的字段，但此时双方的写入权限都将失效（即变得无法进行写入）。

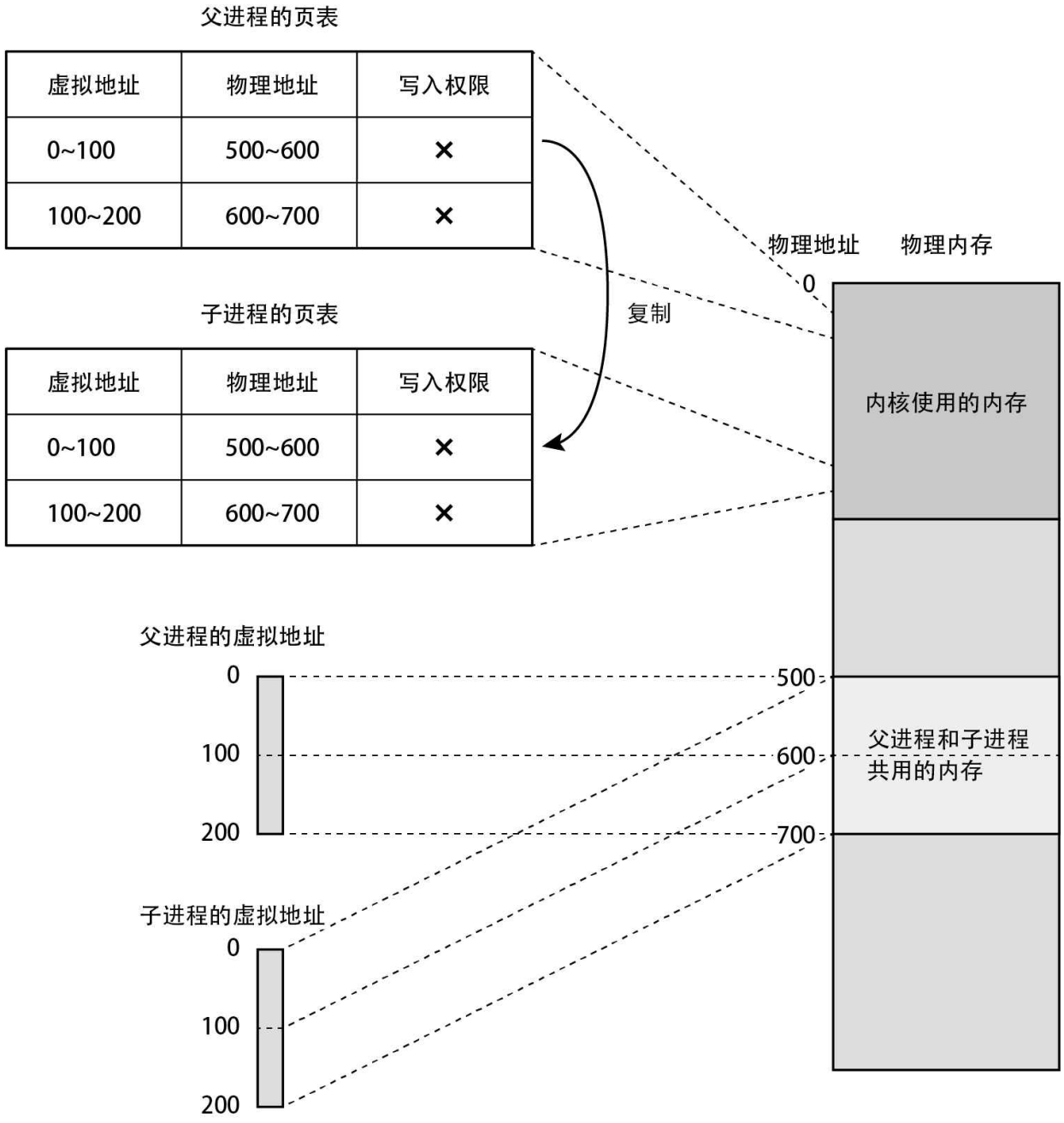

图 5-37　写时复制的运作方式（在调用 fork() 时）

在这之后，假如只进行读取操作，那么父进程和子进程双方都能访问共享的物理页面。但是，当其中一方打算更改任意页面的数据时，则将按照下述流程解除共享。

① 由于没有写入权限，所以在尝试写入时，CPU 将引发缺页中断。

② CPU 转换到内核模式，缺页中断机构开始运行。

③对于被访问的页面，缺页中断机构将复制一份放到别的地方，然后将其分配给尝试写入的进程，并根据请求更新其中的内容。

④为父进程和子进程双方更新与已解除共享的页面对应的页表项。

- 对于执行写入操作的一方，将其页表项重新连接到新分配的物理页面，并赋予写入权限
- 对于另一方，也只需对其页表项重新赋予写入权限即可

子进程在图 5-37 的状态下向地址 100 进行写入时的情形如图 5-38 所示。

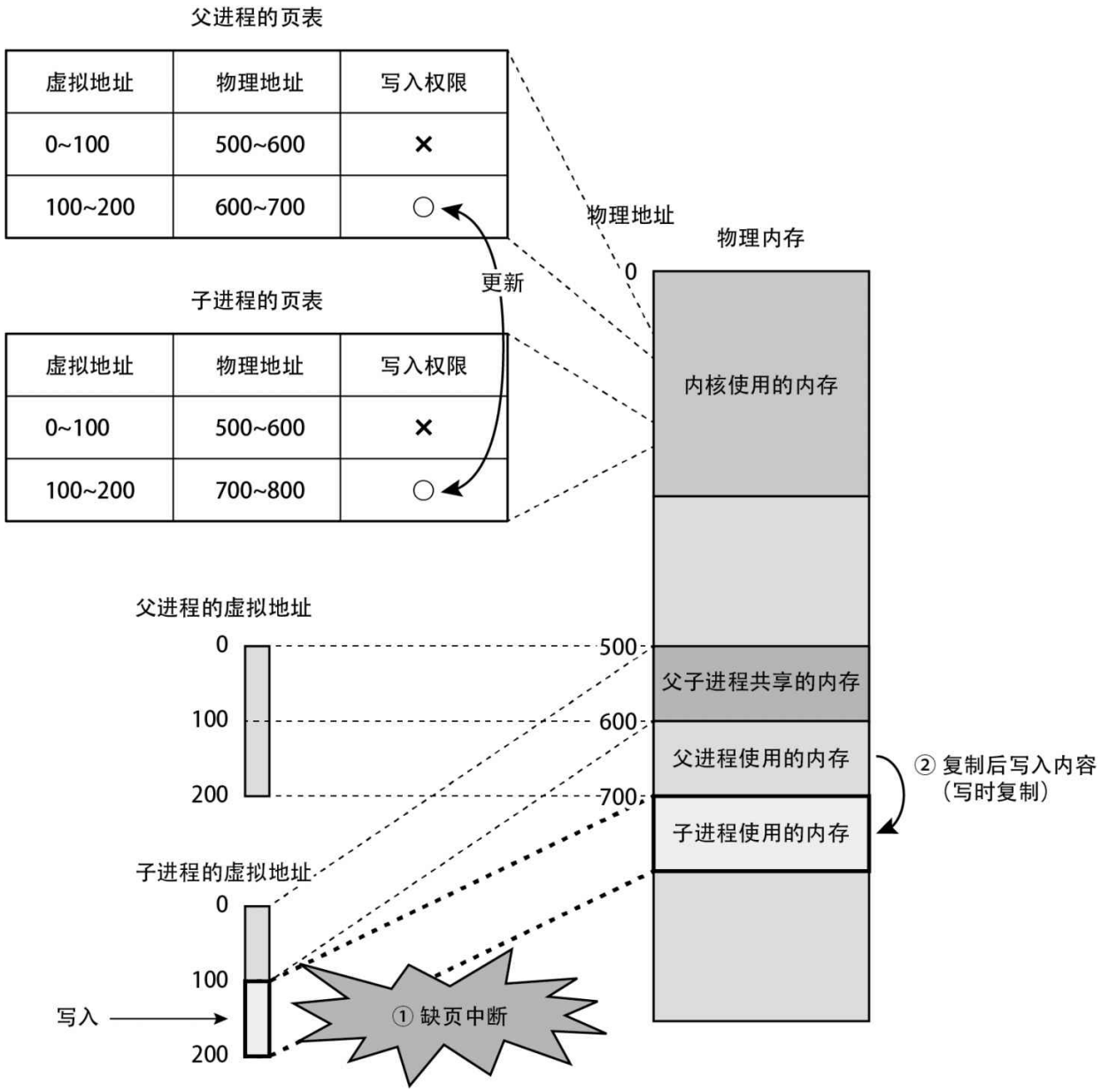

图 5-38　写时复制的运作方式（进行写入时）

在这之后，对于已解除共享关系的页面，父进程和子进程双方都可以自由地进行读写操作。因为物理内存并非在发起 `fork()` 系统调用时进行复制，而是在尝试写入时才进行复制，所以这个机制被称为**写时复制**（Copy on Write，CoW）。

需要注意的是，在写时复制机制下，即便成功调用 `fork()`，如果在写入并引发缺页中断的时间点没有充足的物理页面，也同样会出现物理内存不足的情况。

● 写时复制的实验

接下来，让我们通过实验来观察发生写时复制时的情形。需要确认的事项如下所示。

- 在从调用 **fork()** 到开始写入的这段时间，内存区域是否被父进程和子进程双方共享？
- 在向内存区域执行写入时，是否会引发缺页中断？

为了确认这些事项，需要编写实现下述要求的程序。

① 获取 100 MB 内存，并访问所有页面。

② 确认系统的内存使用量。

③ 调用 **fork()** 系统调用。

④ 父进程和子进程分别执行以下处理。

- 父进程

  i. 等待子进程结束运行。

- 子进程

  i. 显示系统的内存使用量以及自身的虚拟内存使用量、物理内存使用量、硬性页缺失发生次数和软性页缺失发生次数。
  ii. 访问在步骤①中获取的内存区域的所有页面。

  iii. 再次显示系统的内存使用量以及自身的虚拟内存使用量、物理内存使用量、硬性页缺失发生次数和软性页缺失发生次数。

完成后的程序如代码清单 5-6 所示。

代码清单 5-6 cow 程序（cow.c）

```
#include <sys/types.h>
#include <sys/wait.h>
```

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

#define BUFFER_SIZE    (100  1024  1024)
#define PAGE_SIZE      4096
#define COMMAND_SIZE   4096

static char *p
static char command[COMMAND_SIZE];

static void child_fn(char p) {
    printf("** child ps info before memory access ***:\n");
    fflush(stdout);
    snprintf(command, COMMAND_SIZE,
             "ps -o pid,comm,vsz,rss,min_flt,maj_flt | grep '^
             %d'", getpid());
    system(command);
    printf("** free memory info before memory access ***:\n");
    fflush(stdout);
    system("free");

    int i;
    for (i = 0; i < BUFFER_SIZE; i += PAGE_SIZE)
        p[i] = 0;

    printf("*** child ps info after memory access ***:\n");
    fflush(stdout);
    system(command);

    printf("*** free memory info after memory access ***:\n");
    fflush(stdout);
    system("free");

    exit(EXIT_SUCCESS);
}

static void parent_fn(void) {
    wait(NULL);

    exit(EXIT_SUCCESS);
}

int main(void)
{
    char buf;
    p = malloc(BUFFER_SIZE);
    if (p == NULL)
```

```
        err(EXIT_FAILURE, "malloc() failed");

    int i;
    for (i = 0; i < BUFFER_SIZE; i += PAGE_SIZE)
        p[i] = 0;

    printf("** free memory info before fork ***:\n");
    fflush(stdout);
    system("free");

    pid_t ret;
    ret = fork();
    if (ret == -1)
        err(EXIT_FAILURE, "fork() failed");

    if (ret == 0)
        child_fn(p);
    else
        parent_fn();


    err(EXIT_FAILURE, "shouldn't reach here");
}
```

编译并运行这个程序，结果如下。

```
$ cc -o cow cow.c
$ ./cow
*** free memory info before fork ***:
          total     used      free  shared  buff/cache  available
Mem:   32942008  1967716  21552784  298152     9421508   30031664
Swap:         0        0         0
*** child ps info before memory access ***:
12716 cow  106764 102484     27  0
*** free memory info before memory access ***:
          total     used      free  shared  buff/cache  available
Mem:   32942008  1968120  21552380  298152     9421508   30031260
Swap:         0        0         0
*** child ps info after memory access ***:
12716 cow   106764 103432  599   0
*** free memory info after memory access ***:
          total     used      free  shared  buff/cache  available
Mem:   30942008  2071324  21449176  298152     9421508   29928056
Swap:         0        0         0
$
```

根据上面的运行结果，我们可以得知以下两点内容。

- 尽管父进程的内存使用量超过了 100 MB，但从调用 **`fork()`** 到子进程开始往内存写入数据的这段时间，内存使用量仅增加了几百 KB
- 在子进程向内存写入数据后，不但发生缺页中断的次数增加了，系统的内存使用量也增加了 100 MB（这代表内存共享已解除）

关于进程的物理内存使用量，还有一点需要大家注意。

对于共享的内存，父进程和子进程双方会重复计算。因此，所有进程的物理内存使用量的总值会比实际使用量要多。

以实验程序为例，在子进程开始写入数据前，父进程和子进程的实际物理内存使用量共为 100 MB 左右，但双方都会认为自己独占了 100 MB 的物理内存。

5.15 Swap

本章开头提到，当物理内存耗尽时，系统就会进入 OOM 状态。但实际上，Linux 提供了针对 OOM 状态的补救措施，即 Swap 这一利用了虚拟内存机制的功能。

通过这个功能，我们可以将外部存储器的一部分容量暂时当作内存使用。具体来说，在系统物理内存不足的情况下，当出现获取物理内存的申请时，物理内存中的一部分页面将被保存到外部存储器中，从而空出充足的可用内存。这里用于保存页面的区域称为**交换分区** [4]（Swap 分区）。交换分区由系统管理员在构建系统时进行设置。

> [4]虽然有点复杂，但需要注意的是，在 Windows 系统中，交换分区被称为“虚拟内存”。

单靠文字说明可能难以理解，接下来我们将利用图进行说明。

假设系统处于物理内存不足的状态下，且这时需要使用更多的物理内存。在图 5-39 中，进程 B 向尚未关联物理内存的虚拟地址 100 发起访问，这引发了缺页中断。

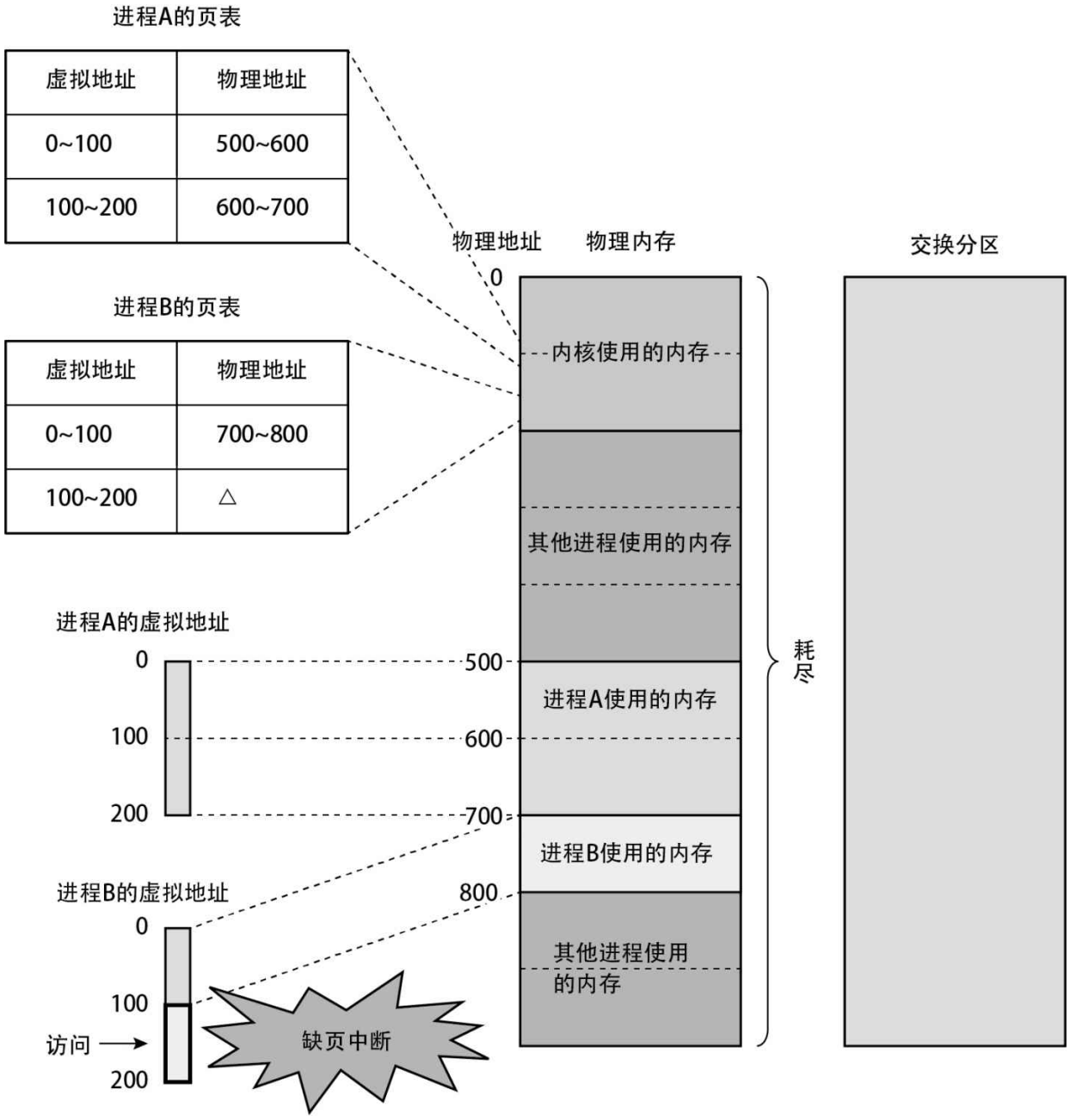

图 5-39　物理内存不足

此时，由于已经没有空闲的物理内存了，所以内核会将正在使用的物理内存中的一部分页面保存到交换分区。这个处理称为**换出**。在图 5-40 中，与进程 A 的虚拟地址 100 ~ 200 对应的物理地址 600 ~ 700 的区域会被换出到交换分区。

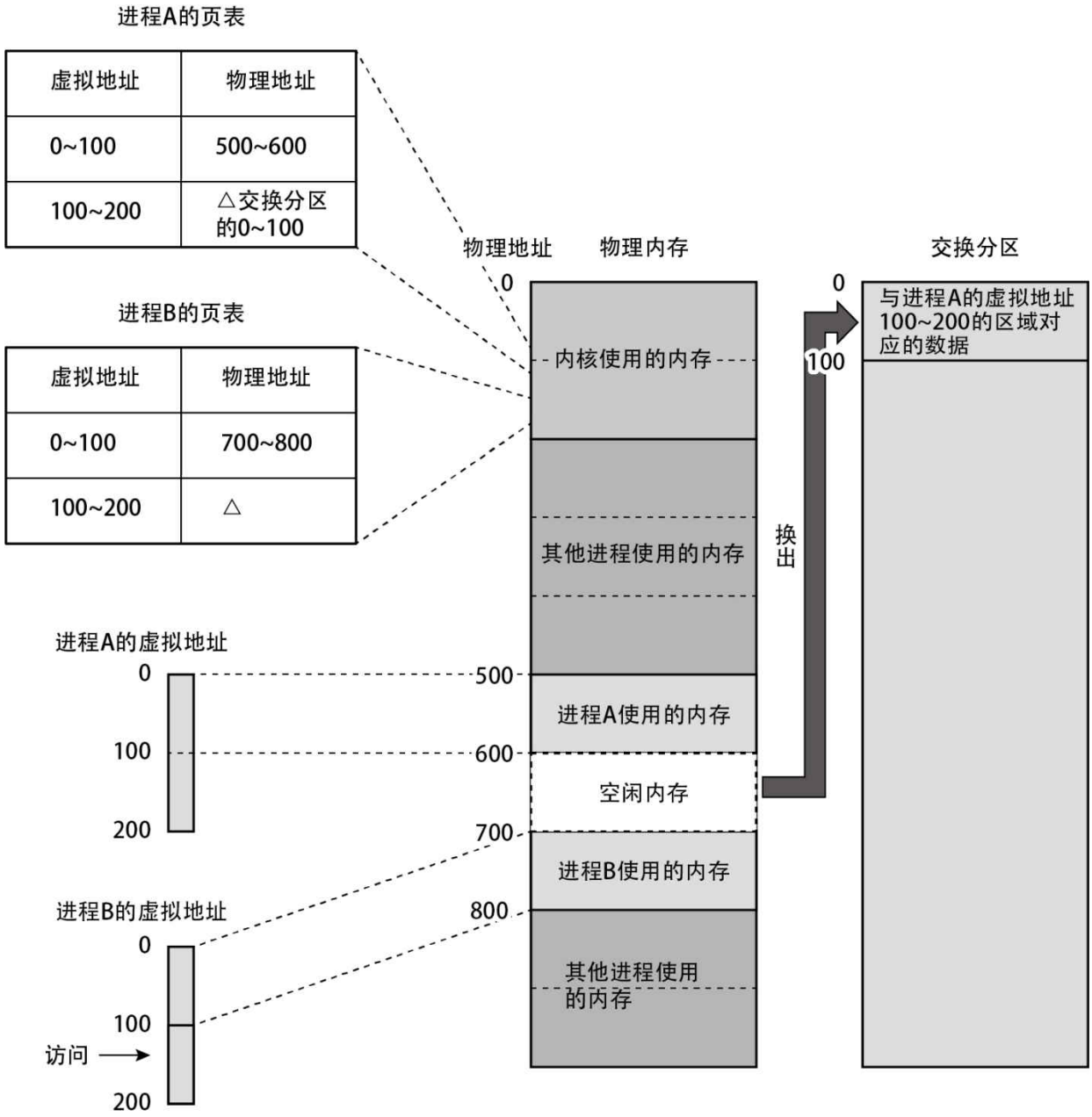

图 5-40　换出

虽然在图 5-40 中，被换出的页面在交换分区上的地址信息记录在页表项中，但实际上是记录在内核中专门用于管理交换分区的区域上的。但是，这方面的内容并非本书的重点，因此在本书中就当作记录在页表项上了。

怎样决定要换出的区域呢？其实是内核基于预设的算法选出的，一般是短时间内应该用不上的区域。但是，我们并不需要深入了解这部分

内容，因此本书不作过多说明。

在通过换出处理空出一块可用内存后，内核将这部分内存分配给进程 B，如图 5-41 所示。

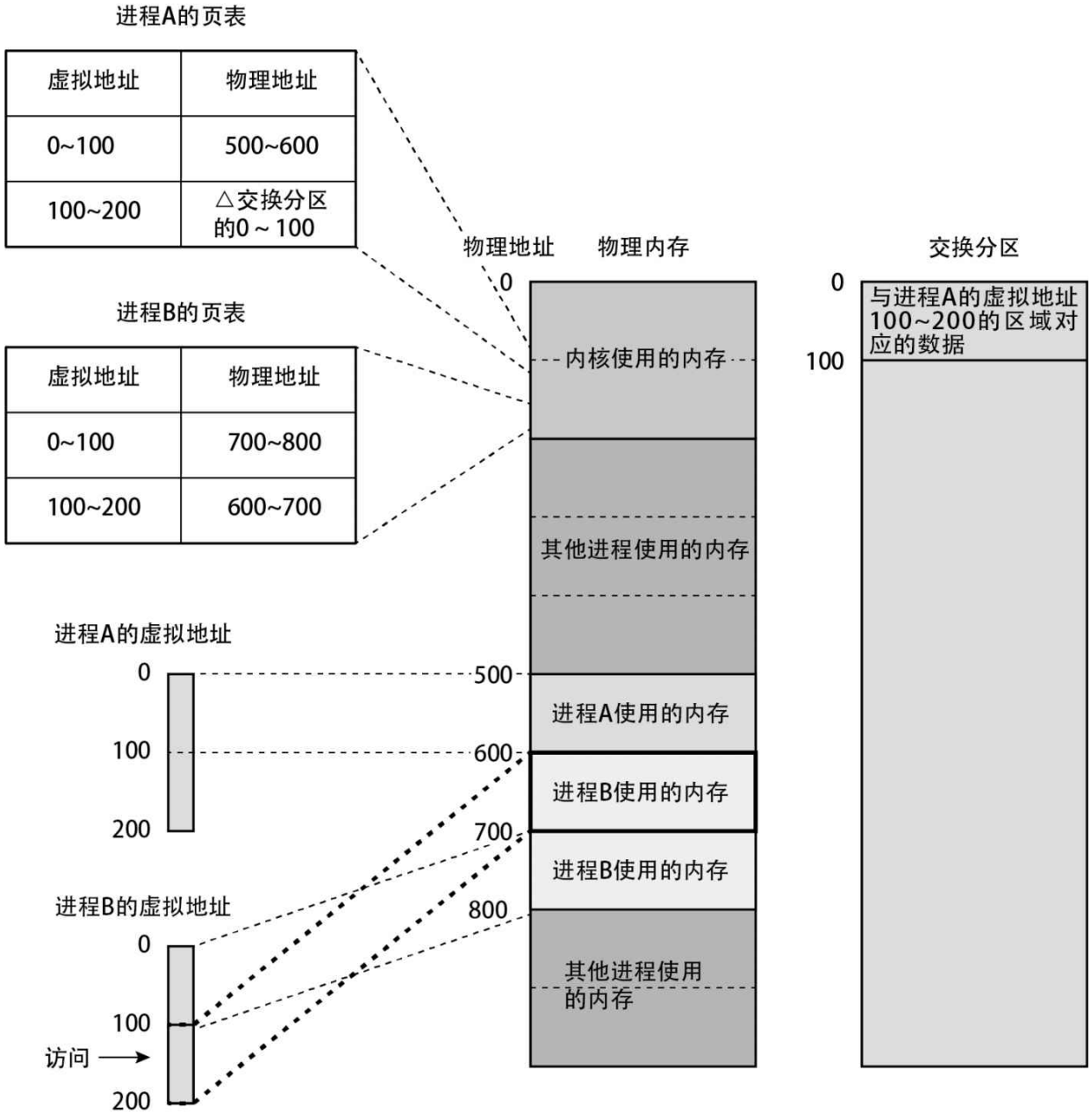

图 5-41　将通过换出处理空出的内存分配给进程 B

假设在经过一段时间后，系统得以空出部分可用内存。在这样的状态下，如果进程 A 对先前保存到交换分区的页面发起访问，就会发生如

图 5-42 所示的情况。

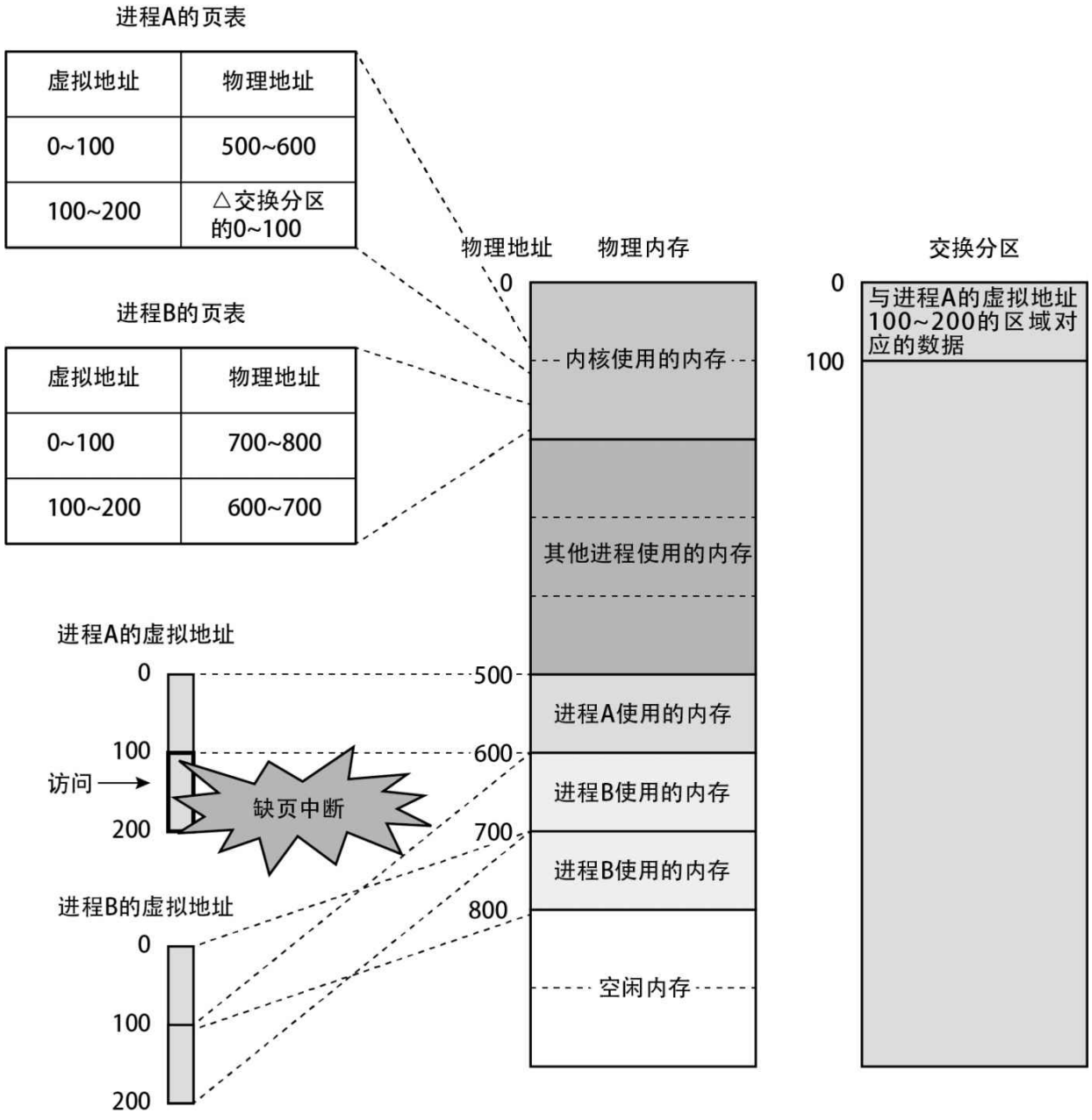

图 5-42　访问已被换出的页面

此时，内核会从交换分区中将先前换出的页面重新拿回到物理内存，这个处理称为**换入**，如图 5-43 所示。