```
*** file contents before overwrite mapped region: hello ←③

*** overwritten mapped region with: HELLO                ←④

$
```

首先，从①所指的内容可知，mmap() 函数成功地把 testfile 文件的数据映射到了地址 0x7fc8cd24d000 上。在成功执行 mmap() 后，内存映射信息中比执行前多出了②所指的信息，该信息表明 testfile 文件成功地映射到了内存上。最后，程序在③那一行输出了更新前的 testfile 文件的内容，在④那一行输出了用于覆写文件内容的数据。

接下来，确认文件内容是否更新成功。

```
$ cat testfile
HELLO
```

看来成功了。通过上面的实验可以看到，即使没有对 testfile 文件使用系统调用 write() 或者 fprintf() 函数，而只是通过 memcpy() 函数把覆写用的数据（overwrite_data 变量）复制到内存映射的区域（file_contents 变量），也同样能更新文件内容。

5.13 请求分页

在上一节，对于创建进程时的内存分配，或者在创建进程后通过 mmap() 系统调用进行的动态内存分配，我们是这样描述它们的流程的。

① 内核直接从物理内存中获取需要的区域。

② 内核设置页表，并关联虚拟地址空间与物理地址空间。

但是，这种分配方式会导致内存的浪费。因为在获取的内存中，有一部分内存在获取后，甚至直到进程运行结束都不会使用，例如：

- 用于大规模程序中的、程序运行时未使用的功能的代码段和数据段
- 由 **glibc** 保留的内存池中未被用户利用的部分

为了解决这个问题，Linux 利用请求分页机制来为进程分配内存。

在请求分页机制中，对于虚拟地址空间内的各个页面，只有在进程初次访问页面时，才会为这个页面分配物理内存。页面的状态除了前面提到过的“未分配给进程”与“已分配给进程且已分配物理内存”这两种以外，还存在“已分配给进程但尚未分配物理内存”这种状态。只看文字可能难以理解，接下来，我们通过图来说明一下请求分页的处理流程。

首先，在创建进程时，在其虚拟地址空间中的与代码段和数据段对应的页面上，添加“已为进程分配该区域（页面）”这样的信息[2]，但暂时不会分配物理内存，如图 5-30 所示。图 5-30 中的△表示“已分配给进程但尚未分配物理内存”的状态。

[2]实际上该信息并非保存在页表上，这里为了方便说明而假设它保存在页表上。

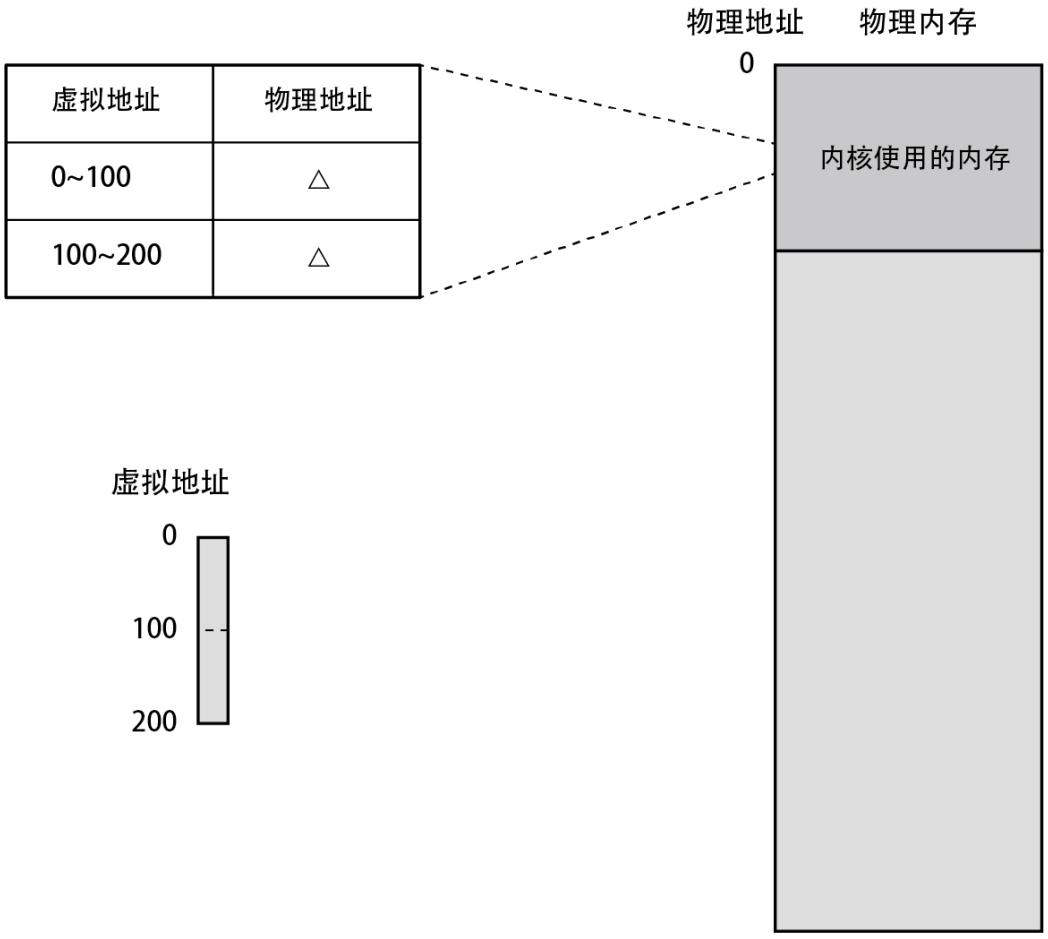

图 5-30　刚创建完进程时（未分配物理内存）

然后，在从入口点开始运行进程时，为入口点所属的页面分配物理内存，如图 5-31 所示。

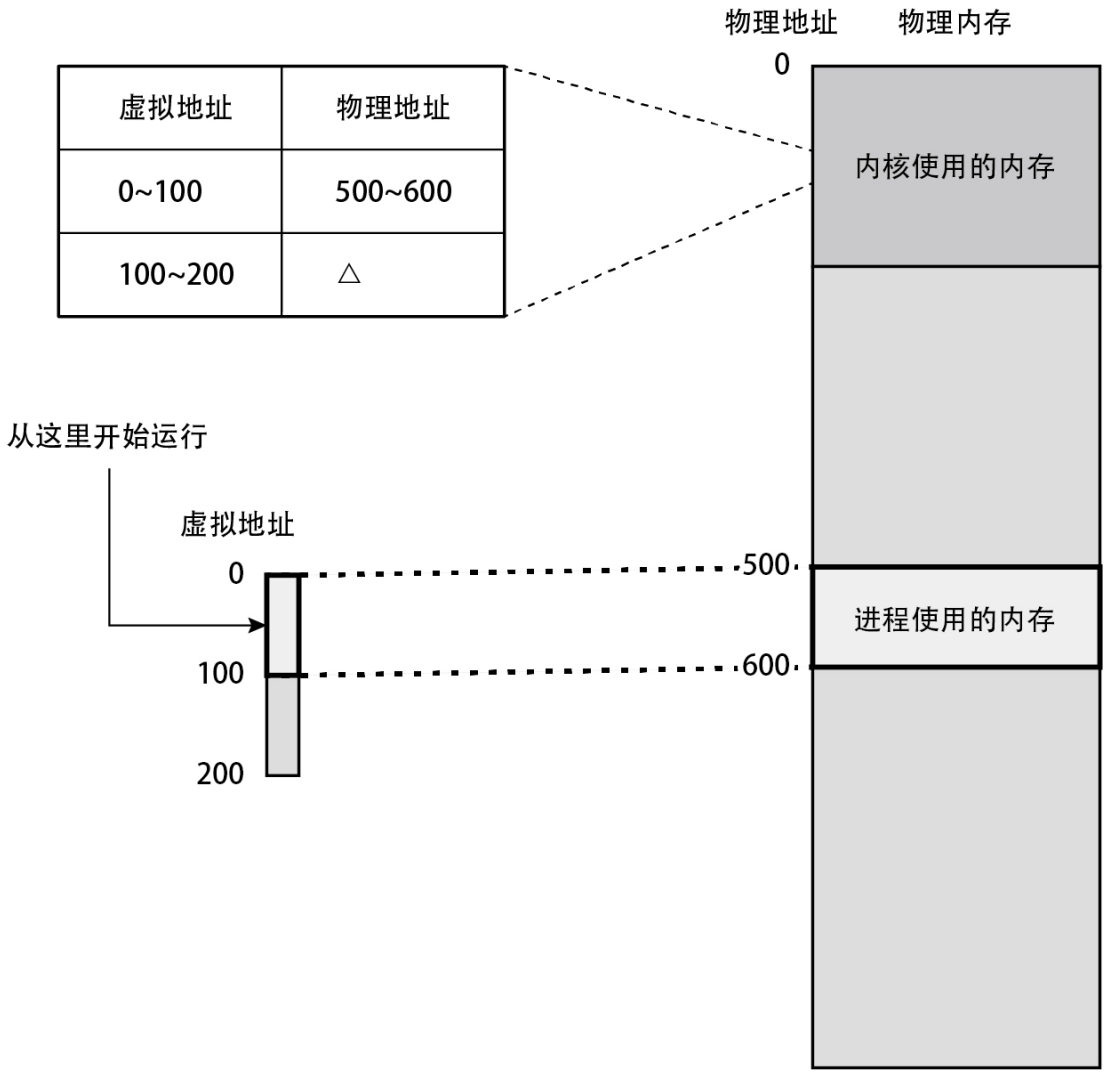

图 5-31　在开始运行时，为入口点所属的页面分配内存

此时的处理流程如下所示。

① 进程访问入口点。

② CPU 参照页表，筛选出入口点所属的页面中哪些虚拟地址未关联物理地址。

③ 在 CPU 中引发缺页中断。

④ 内核中的缺页中断机构为步骤①中访问的页面分配物理内存，并更新其页表。

⑤ 回到用户模式，继续运行进程。

另外，进程并不会感知到自身在运行时曾发生过缺页中断。

此后，每当访问新的区域时，都如上述流程所示，先触发缺页中断，然后分配物理内存，并更新对应的页表（图 5-32）。

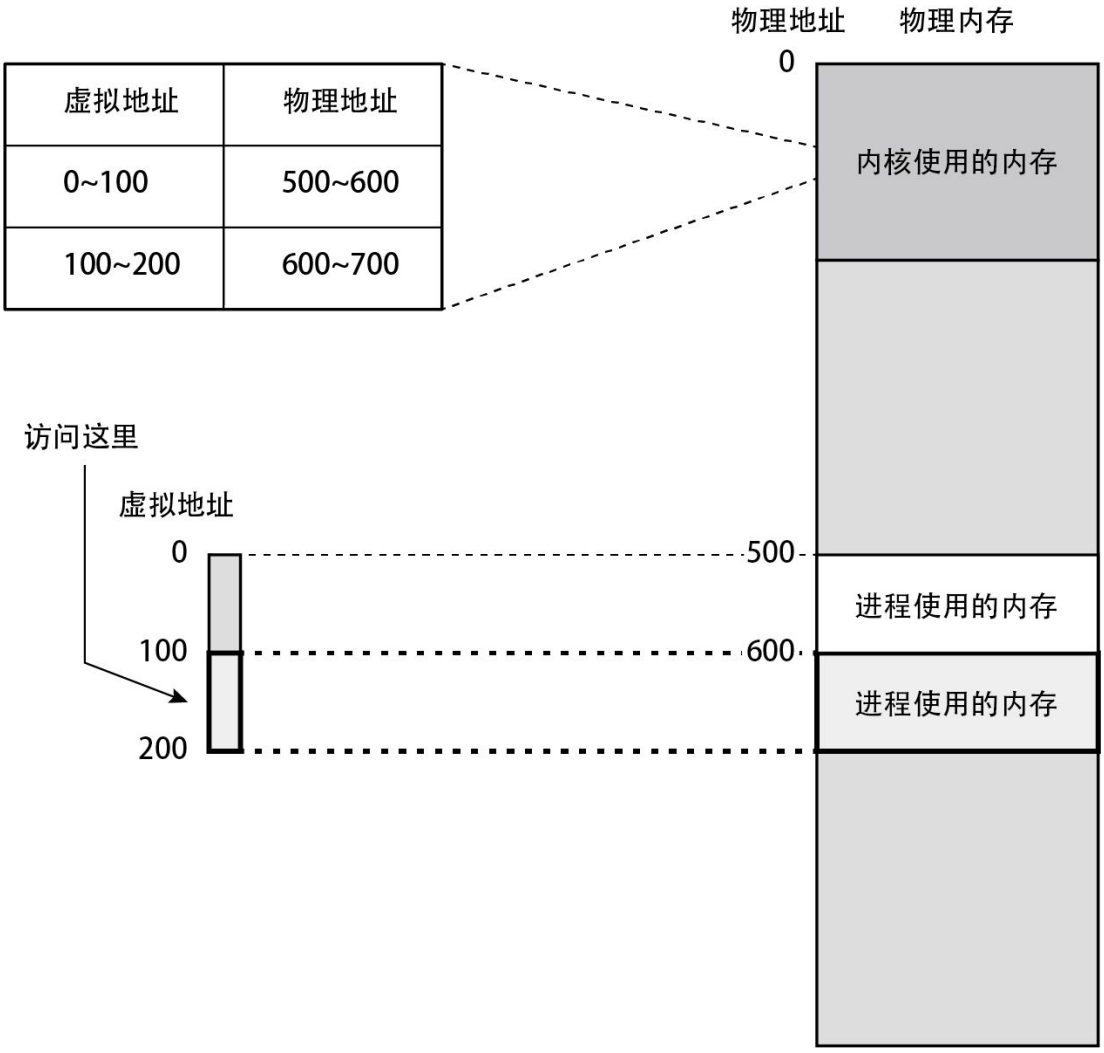

图 5-32　分配物理内存

图 5-33 所示为进程通过 mmap() 函数动态获取内存时的情形。

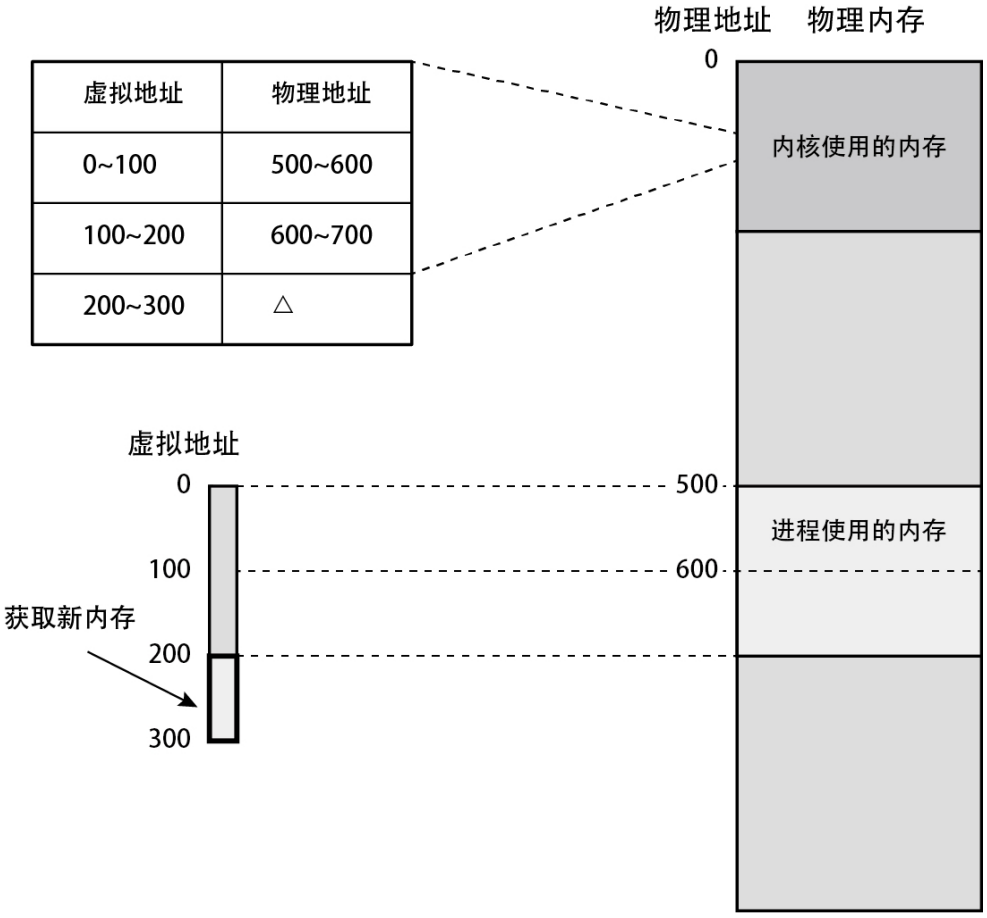

图 5-33　通过 mmap() 函数动态获取内存（未分配物理内存）

在分配成功后，如果对该内存发起访问，进程就会为其分配物理内存，如图 5-34 所示。

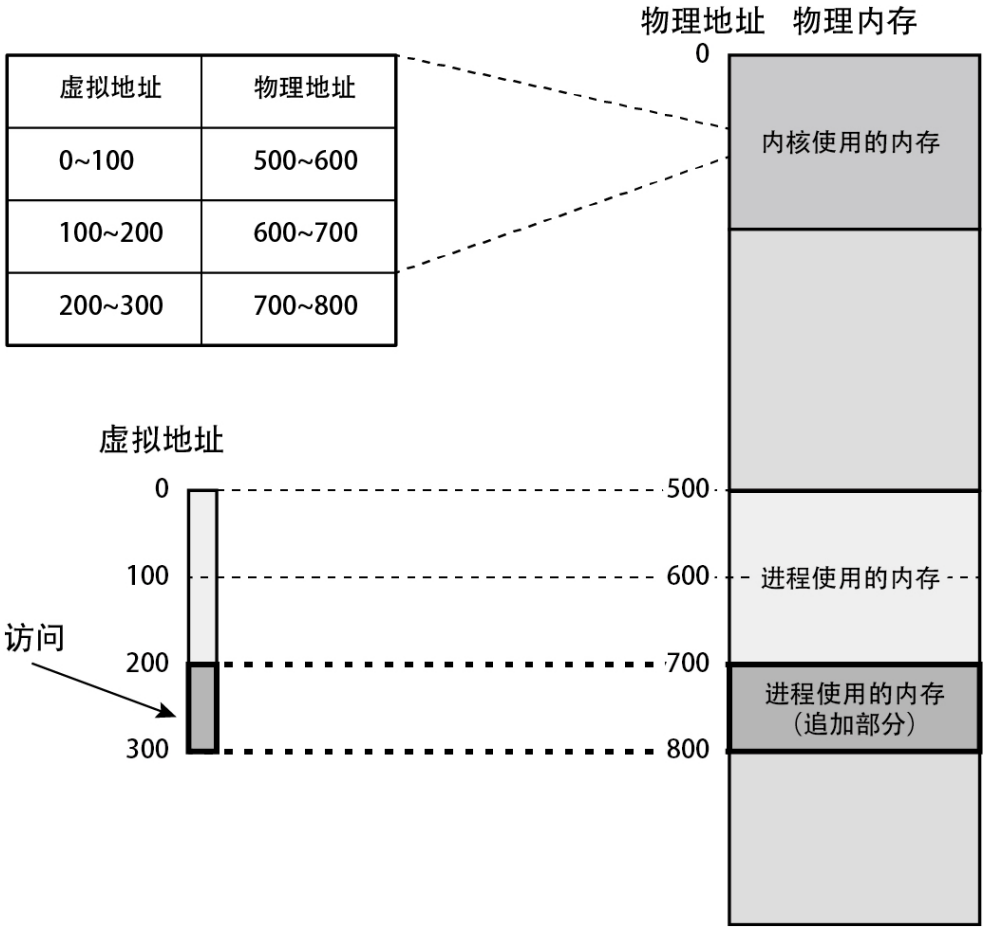

图 5-34　在访问已获取的内存时，为其分配物理内存

我们将“进程通过 mmap() 函数等成功获取内存”表述为“成功获取虚拟内存”，将“访问所获取的虚拟内存并将虚拟内存关联到物理内存”表述为“成功获取物理内存”。

在通过请求分页机制获取内存时，无论 mmap() 函数的调用成功与否，在向内存写入数据时，如果物理内存中已经没有充足的可用内存，就会引发物理内存不足的问题。

● 请求分页的实验

下面，让我们一起来观察一下发生请求分页时的情形。需要确认的事项如下所示。

- 在获取内存后，是否只会增加虚拟内存使用量，而不会增加物理内存使用量？
- 在访问已获取的内存时，物理内存使用量是否会增加，与此同时是否会发生缺页中断？

为了确认这些事项，需要编写实现下述要求的程序。

- 处理流程如下所示

  ① 输出一条信息，用于提示尚未获取内存，随后等待用户按下 Enter 键。

  ② 获取 100 MB 的内存。

  ③ 输出一条信息，用于提示成功获取内存，随后等待用户按下 Enter 键。

  ④ 从头到尾逐页访问已获取的内存，每访问 10 MB 内存，就输出一条信息，用于提示当前访问进度。

  ⑤ 在访问完所有在步骤②中获取的内存后，输出相应的提示信息，随后等待用户按下 Enter 键。

- 在每条信息的开头添加时间戳

完成后的程序如代码清单 5-4 所示。

代码清单 5-4　demand-paging 程序（demand-paging.c）

```
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <err.h>
#define BUFFER_SIZE   (100  1024  1024)
#define NCYCLE        10
#define PAGE_SIZE     4096

int main(void)
{
```

```
    char p;
    time_t t;
    char s;

    t = time(NULL);
    s = ctime(&t);
    printf("%.*s: before allocation, please press Enter key\n",
           (int)(strlen(s) - 1), s);
    getchar();

    p = malloc(BUFFER_SIZE);
    if (p == NULL)
        err(EXIT_FAILURE, "malloc() failed");

    t = time(NULL);
    s = ctime(&t);
    printf("%.*s: allocated %dMB, please press Enter key\n",
           (int)(strlen(s) - 1), s, BUFFER_SIZE / (1024  1024));
    getchar();

    int i;
    for (i = 0; i < BUFFER_SIZE; i += PAGE_SIZE) {
        p[i] = 0;
        int cycle = i  (BUFFER_SIZE  NCYCLE);
        if (cycle != 0 && i % (BUFFER_SIZE / NCYCLE) == 0) {
            t = time(NULL);
            s = ctime(&t);
            printf("%.s: touched %dMB\n",
                   (int) (strlen(s) - 1), s, i / (1024*1024));
            sleep(1);
        }
    }

    t = time(NULL);
    s = ctime(&t);
    printf("%.*s: touched %dMB, please press Enter key\n",
           (int) (strlen(s) - 1), s, BUFFER_SIZE / (1024 * 1024));
    getchar();

    exit(EXIT_SUCCESS);
}
```

编译并运行这个程序，结果如下。

```
$ cc -o demand-paging demand-paging.c
$ ./demand-paging
Mon Dec 25 22:06:15 2017: before allocation. Please press ↵
Enter key
```

```
Mon Dec 25 22:06:18 2017: allocation 100MB. Please press ↵
Enter key

Mon Dec 25 22:06:21 2017: touched 10MB
Mon Dec 25 22:06:22 2017: touched 20MB
Mon Dec 25 22:06:23 2017: touched 30MB
Mon Dec 25 22:06:24 2017: touched 40MB
Mon Dec 25 22:06:25 2017: touched 50MB
Mon Dec 25 22:06:26 2017: touched 60MB
Mon Dec 25 22:06:27 2017: touched 70MB
Mon Dec 25 22:06:28 2017: touched 80MB
Mon Dec 25 22:06:29 2017: touched 90MB
Mon Dec 25 22:06:30 2017: touched 100MB. Please press ↵
Enter key

$
```

虽然程序能顺利运行，但只看上面的运行结果是无法得知任何信息的。为了令这个 demand-paging 程序有用武之地，必须在另外一个终端上运行采集系统信息的程序。

首先，在一个终端上运行 demand-paging 程序，在另一个终端上通过 sar -r 命令每秒采集一次系统的内存信息。运行程序的终端输出了与刚才同样的内容。

```
$ ./demand-paging
Mon Dec 25 22:07:43 2017: before allocation. Please press ↵
Enter key

Mon Dec 25 22:07:45 2017: allocation 100MB. Please press ↵
Enter key
Mon Dec 25 22:07:47 2017: touched 10MB
Mon Dec 25 22:07:48 2017: touched 20MB
Mon Dec 25 22:07:49 2017: touched 30MB
Mon Dec 25 22:07:50 2017: touched 40MB
Mon Dec 25 22:07:51 2017: touched 50MB
Mon Dec 25 22:07:52 2017: touched 60MB
Mon Dec 25 22:07:53 2017: touched 70MB
Mon Dec 25 22:07:54 2017: touched 80MB
Mon Dec 25 22:07:55 2017: touched 90MB
Mon Dec 25 22:07:56 2017: touched 100MB. Please press ↵
Enter key

$
```

另外一个终端输出的内容如下所示。

```
$ sar -r 1
it...
22:07:41 kbmemfree  kbmemused %memused kbbuffers  kbcached ↲
kbcommit   %commit   kbactive  kbinact   kbdirty
22:07:42  21699640   11242368    34.13      4676   8818284 ↲
8072548      24.51    8865224  1550120       160
22:07:43  21699888   11242120    34.13      4676   8818280 ↲
8072548      24.51    8865052  1550120       160  ←获取内存前
22:07:44  21699888   11242120    34.13      4676   8818284 ↲
8072548      24.51    8865116  1550124       160  ←获取内存后
22:07:45  21698896   11243112    34.13      4676   8818284 ↲
8072732      24.81    8866228  1550120       160
22:07:46  21699516   11242492    34.13      4676   8818284 ↲
8072732      24.81    8865360  1550120       160
22:07:47  21688576   11253432    34.16      4676   8818280 ↲
8072732      24.81    8877448  1550116       160
22:07:48  21677772   11264236    34.19      4676   8818280 ↲
8072732      24.81    8887760  1550116       160
22:07:49  21667400   11274608    34.23      4676   8818284 ↲
8072732      24.81    8897840  1550120       160
22:07:50  21657160   11284848    34.26      4676   8818284 ↲
8072732      24.81    8908320  1550120       160
22:07:51  21646796   11295212    34.29      4676   8818316 ↲
8072732      24.81    8918672  1550120       180
22:07:52  21636432   11305576    34.32      4676   8818320 ↲
8072732      24.81    8928956  1550140       188
22:07:53  21626200   11315808    34.35      4676   8818316 ↲
8072020      24.81    8939468  1550132       188
22:07:54  21614996   11327012    34.38      4676   8818320 ↲
8072020      24.81    8950384  1550136       188
22:07:55  21604368   11337640    34.42      4676   8818324 ↲
8072020      24.81    8960768  1550140       188
22:07:56  21596176   11345832    34.44      4676   8818324 ↲
8072020      24.81    8968980  1550140       188
                                ←到这里为止保持每秒获取10 MB 内存
22:07:57  21596192   11345816    34.44      4676   8818324 ↲
8072020      24.81    8968504  1550140       192
22:07:58  21698032   11243976    34.13      4676   8818328 ↲
8060456      24.47    8866964  1550140       192
                                                  ←进程运行结束
( 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。

- 即使已经获取内存区域，在访问这个区域的内存前，系统上的物理内存使用量（**kbmemused** 字段的值）也几乎 [3]不会发生改变

- 在开始访问内存后，内存使用量每秒增加 10 MB 左右
- 在访问结束后，内存使用量不再发生变化
- 在进程结束运行后，内存使用量回到开始运行进程前的状态

[3]这里使用“几乎”是因为内存使用量还受系统上正在运行的其他程序或内核的行为的影响。

接下来是同样的做法，在一个终端上运行程序，在另一个终端上通过 `sar -B` 命令每秒观测一次缺页中断的发生情况。

```
$ sar -B 1
( 略 )
22:13:05   pgpgin/s  pgpgout/s  fault/s  majflt/s  pgfree/s ↲
pgscank/s pgscand/s  pgsteal/s   %vmeff
22:13:06       0.00       0.00     2.00      0.00     33.00 ↲
0.00           0.00       0.00     0.00
22:13:07       0.00       0.00     1.00      0.00     73.00 ↲
0.00           0.00       0.00     0.00
22:13:08       0.00       0.00     0.00      0.00     18.00 ↲
0.00           0.00       0.00     0.00
22:13:09       0.00       0.00   338.00      0.00     35.00 ↲
0.00           0.00       0.00     0.00      ←开始访问内存
22:13:10       0.00       0.00     5.00      0.00     18.00 ↲
0.00           0.00       0.00     0.00
22:13:11       0.00       0.00    30.69      0.00    268.32 ↲
0.00           0.00       0.00     0.00
22:13:12       0.00       0.00    31.00      0.00     60.00 ↲
0.00           0.00       0.00     0.00
22:13:13       0.00       4.00    35.00      0.00     49.00 ↲
0.00           0.00       0.00     0.00
22:13:14       0.00       0.00     5.00      0.00     17.00 ↲
0.00           0.00       0.00     0.00
22:13:16       0.00       0.00    31.00      0.00     62.00 ↲
0.00           0.00       0.00     0.00
22:13:17       0.00       0.00    31.00      0.00     44.00 ↲
0.00           0.00       0.00     0.00
22:13:18       0.00       0.00    31.00      0.00     61.00 ↲
0.00           0.00       0.00     0.00
22:13:19       0.00       0.00   293.00      0.00    119.00 ↲
0.00           0.00       0.00     0.00      ←内存访问结束
22:13:20       0.00       0.00     0.00      0.00     34.00 ↲
0.00           0.00       0.00     0.00
( 略 )
```

可以看到，在进程访问所获取的内存区域这段时间，表示每秒发生的缺页中断次数的 `fault/s` 字段的值有所增加。

还有一点没在上面的例子中体现出来，那就是即使进程再次访问同一个内存区域，也不会再次引发缺页中断。因为物理内存已经在第一次访问时完成分配。想亲自确认这一点的读者，可以更改源代码尝试一下。

下面，我们先不管系统整体的统计信息，将注意力集中在各个进程的统计信息上。

需要确认的信息为虚拟内存量、已分配的物理内存量，以及在创建进程后发生缺页中断的总次数。这些数值分别通过 ps -eo 命令中的 vsz、rss、maj_flt 以及 min-flt 获取。

执行如代码清单 5-5 所示的脚本，每秒输出一次该命令的值。

代码清单 5-5　vsz-rss 脚本（vsz-rss.sh）

```
#!binbash

while true ; do
    DATE=$(date | tr -d '\n')
    INFO=$(ps -eo pid,comm,vsz,rss,maj_flt,min_flt | grep
           demand-paging | grep -v grep)
    if [ -z "$INFO" ] ; then
        echo "$DATE: target process seems to be finished"
        break
    fi
    echo "${DATE}: ${INFO}"
    sleep 1
done
```

vsz-rss 脚本的运行结果如下所示。各行中位于 demand-paging 字段右侧的 4 个字段分别是虚拟内存量、已分配的物理内存量、硬性页缺失发生次数以及软性页缺失发生次数。

```
$ ./vsz-rss.sh
Mon Dec 25 22:18:27 JST 2017: 11455 demand-paging     4356 ↲
648      0     82
Mon Dec 25 22:18:28 JST 2017: 11455 demand-paging     4356 ↲
648      0     82
Mon Dec 25 22:18:29 JST 2017: 11455 demand-paging     4356 ↲
648      0     82
Mon Dec 25 22:18:30 JST 2017: 11455 demand-paging   106760 ↲
648      0     83
```

```
                                                          ←获取内存
Mon Dec 25 22:18:31 JST 2017: 11455 demand-paging   106760 ↲
648      0     83
Mon Dec 25 22:18:32 JST 2017: 11455 demand-paging   106760 ↲
648      0     83
Mon Dec 25 22:18:33 JST 2017: 11455 demand-paging   106760 ↲
12596    0    352
                                                      ←开始访问内存
Mon Dec 25 22:18:34 JST 2017: 11455 demand-paging   106760 ↲
22836    0    357
Mon Dec 25 22:18:36 JST 2017: 11455 demand-paging   106760 ↲
33076    0    362
Mon Dec 25 22:18:37 JST 2017: 11455 demand-paging   106760 ↲
43316    0    367
Mon Dec 25 22:18:38 JST 2017: 11455 demand-paging   106760 ↲
53556    0    372
Mon Dec 25 22:18:39 JST 2017: 11455 demand-paging   106760 ↲
63796    0    377
Mon Dec 25 22:18:40 JST 2017: 11455 demand-paging   106760 ↲
74036    0    382
Mon Dec 25 22:18:41 JST 2017: 11455 demand-paging   106760 ↲
84276    0    387
Mon Dec 25 22:18:42 JST 2017: 11455 demand-paging   106760 ↲
94516    0    392
Mon Dec 25 22:18:43 JST 2017: 11455 demand-paging   106760 ↲
103628   0    644
                                                      ←内存访问结束
Mon Dec 25 22:18:44 JST 2017: 11455 demand-paging   106760 ↲
103628   0    644
Mon Dec 25 22:18:45 JST 2017: target process seems to be  ↲
finished
```

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。

- 在已获取内存但尚未进行访问这段时间内，虚拟内存量比获取前增加了约 100 MB，但物理内存量并没有发生变化
- 在开始访问内存后，物理内存量每秒增加 10 MB 左右，但虚拟内存量没有发生变化
- 在访问结束后，物理内存量比开始访问前多了约 100 MB

通过这一连串的实验，相信大家应该对请求分页机制有了大致的了解。

● 虚拟内存不足与物理内存不足

我们都知道，在进程运行时，如果获取内存失败，进程就会异常终止。但不知大家是否知道，获取内存失败也分为虚拟内存不足与物理内存不足两种情况。

当进程把虚拟地址空间的范围内的虚拟内存全部获取完毕后，就会导致虚拟内存不足。举个例子，在虚拟地址空间的大小只有 500 字节的情况下，图 5-35 中的情况就会引发虚拟内存不足。由于已经使用完了全部虚拟地址空间，所以即使尚有 300 字节的可用物理内存，也会引发虚拟内存不足。

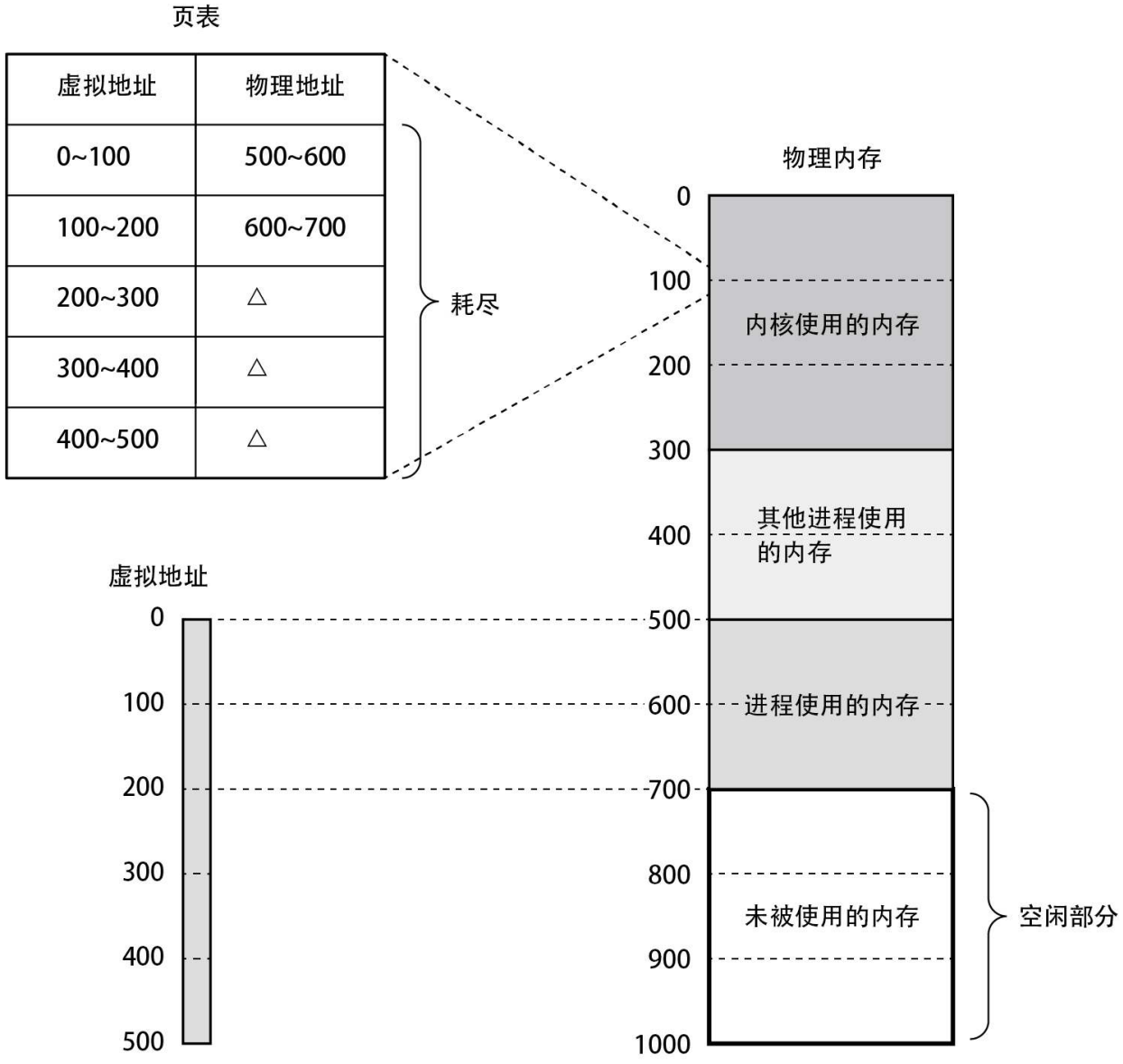

图 5-35　虚拟内存不足