

内存管理

Linux 通过 **内存管理系统** 管理内存，除了各种进程以外，内核本身也需要内存。

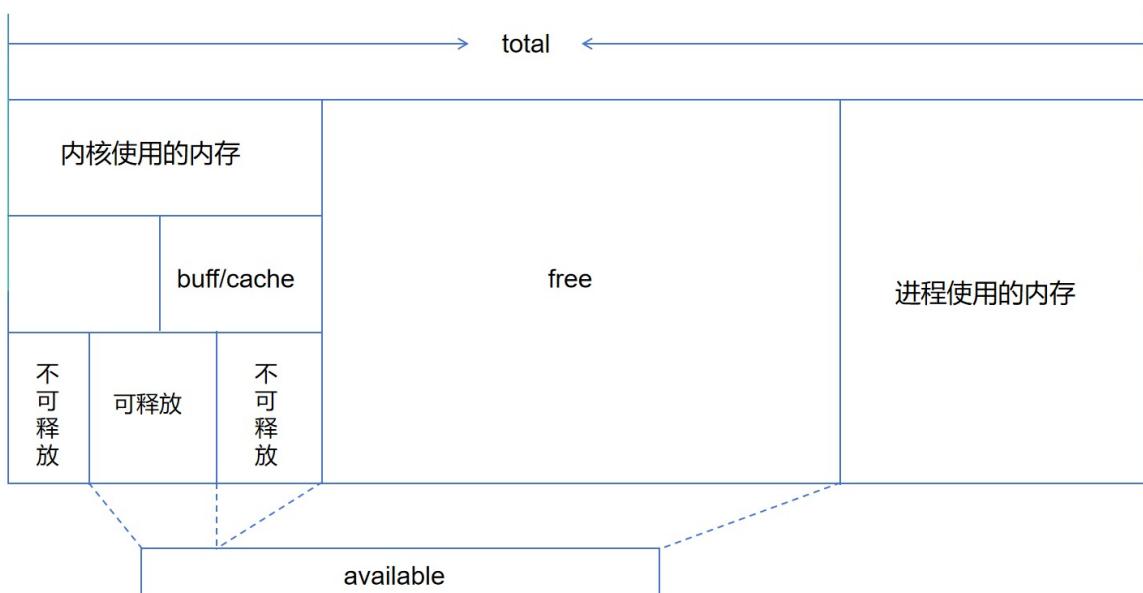
1. 内存相关的统计信息

使用 `free` 命令，得到：

```
syz@syz:~$ free
total        used        free      shared  buff/cache   available
Mem:    7942512     524232    7075648          3300    342632    7183460
Swap:  2097152          0    2097152
syz@syz:~$
```

- `total`：系统搭载的物理内存总量（约 **7.9GB**）。
- `free`：表面上可用内存量（约 **7.0GB**）。
- `buff/cache`：缓冲区缓存与页面缓存（约 **342 MB**）。
- `available`：实际可用内存量。本字段的值为 `free` 字段的值加上内存不足，内核可释放的内存量。“可释放的内存”指缓冲区缓存与页面缓存中的大部分内存，以及内核除此以外的用于其他地方的部分内存（约 **7.1GB**）。

大致为：



同时 `sar -r 1` 可以采集一些内存的相关信息：

```

syz@syz:~$ sar -r 1
Linux 5.15.133.1-microsoft-standard-WSL2 (syz) 02/26/24      _x86_64_      (20 CPU)

18:39:14    kbmemfree   kbavail  kbmemused %memused kbbuffers  kbcached  kbcommit %commit  kbactive  kbinact   kbdirty
18:39:15    7095492    7204132  439668   5.54     14780    296628   733864   7.31     94588    389104   3004
18:39:16    7095492    7204132  439668   5.54     14788    296620   733864   7.31     94588    389104   3004
18:39:17    7095492    7204132  439668   5.54     14788    296620   733864   7.31     94588    389104   3004
18:39:18    7095492    7204268  439524   5.53     14786    296764   730520   7.28     94592    388860   3020
^C
Average:   7095492    7204166  439632   5.54     14786    296658   733028   7.30     94589    389043   3008
syz@syz:~$ █

```

2. 内存不足

随着内存使用量增加，可用内存变得越来越少，首先内核会将可释放的内存释放出来：



如果内存使用量继续增加，系统就会陷入做什么都缺乏足够的内存，以至于陷入 **内存不足** (Out of memory)。此时内存管理系统会运行 **OOM killer** 的功能，它会选出合适的进程，并将其 **kill** 掉，以释放更多内存。

个人电脑出现 OOM 问题不大，但是如果在商用的系统服务器上出现，则完全不知道 OOM killer 杀掉了哪个程序，因此也有将服务器 `sysctl` 的 `vm.panic_on_oom` 参数从默认的 `0` 调整为 `1`，这样使得发生 OOM 时候强制系统关机。

3. 简单的内存分配

本小节，暂时先不谈虚拟内存部分，只讨论简单的分配制度，内核为进程分配内存的时机大致分为以下两种：

- 在创建进程时 (**fork, fork and exec**)
- 在创建完进程后，动态分配内存时

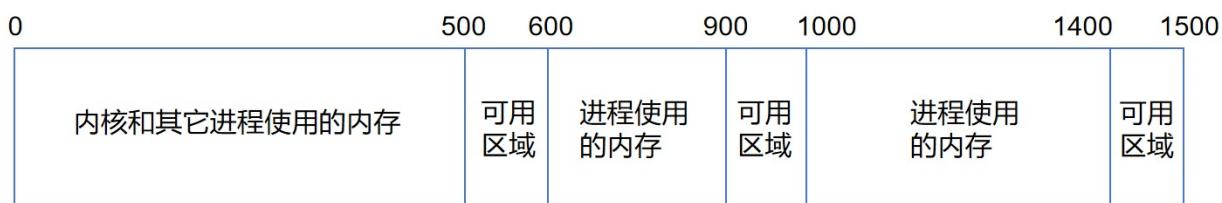
进程被创建后，如果需要更多内存，进程向内核发出用于获取内存的系统调用，提出分配内存的请求。内核收到请求后，会按照请求量在可用内存中分出相应大小的内存，并将这部分内存的起始地址返回给提出请求的进程。·

但是这种模式会引起下列问题：

- 内存碎片化。
- 访问用于其它用途的内存区域。
- 难以执行多任务。

3.1 内存碎片化

例如：



假设上述地址单位为 KB。上述现象(内存碎片化)会导致本来还剩 **300KB** 的内存，却分出 **300KB** 的内存

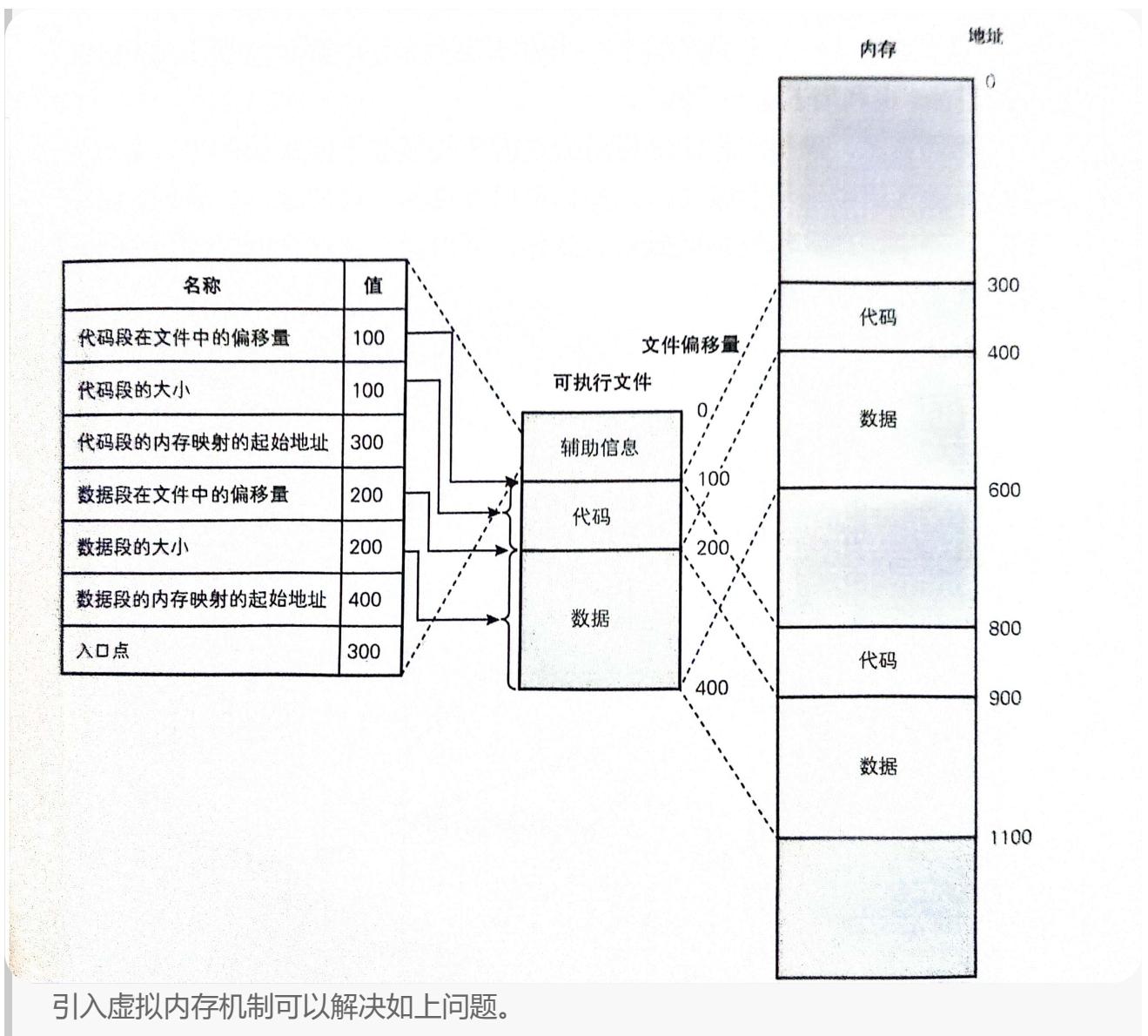
3.2 访问用于其它用途的内存

上述模式中，进程均可以通过内存地址来访问内核和其它进程所使用的内存，因此容易存在数据被泄露或者损毁的风险，如果内核的数据被损毁，系统将无法运行。

3.3 难以执行多任务

之前我们了解到可执行程序存在一个固定的代码块，数据块的存储起始地址，那么如果计算机按照之前我们提及的模式管理内存，如果同时运行两个相同程序，按道理这两个程序都需要在同一块，这就意味着只能运行一个，然而事实上我们是可以运行多个程序的，另一种放置方式如图所示：

但是此时第 2 个程序由于代码和数据指向的内存地址与预期不同而无法执行。



引入虚拟内存机制可以解决如上问题。

4. 虚拟内存

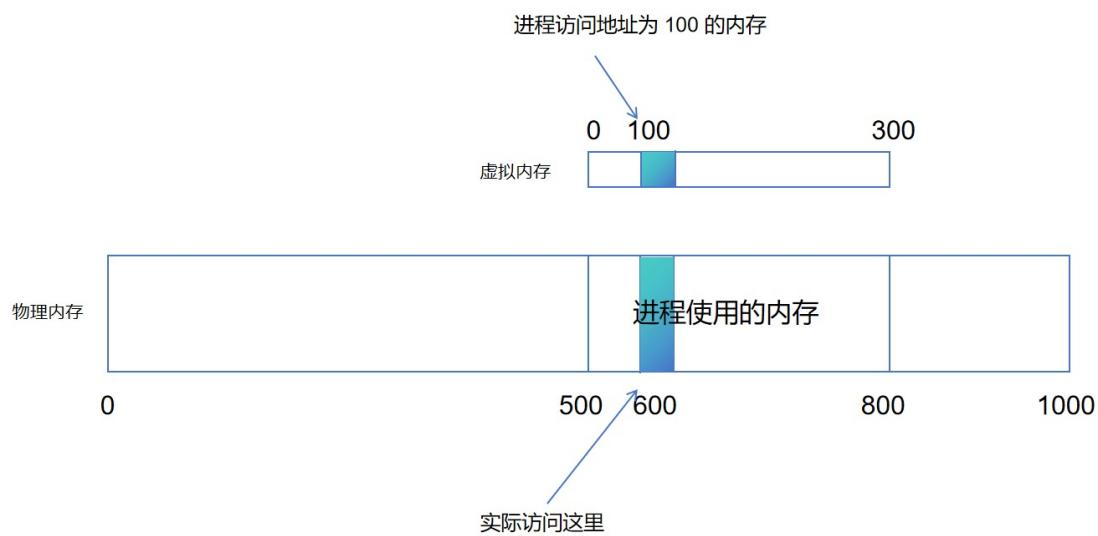
现代 CPU 搭载 **虚拟内存** 的功能，它使得进程无法直接访问系统搭载的内存，而是通过虚拟地址间接访问，系统搭载的内存的实际地址被称为 **物理地址**，可以通过地址访问的范围称为 **地址空间**

之前学到的通过 `readelf` 或者 `cat /proc/[pid]/maps` 输出的地址是 **虚拟地址**。进程无法访问真实的内存，不存在直接访问物理地址的方法。

4.1 页表

通过保存放置于内核使用的内存的页表，可以完成虚拟地址到物理地址的转换。虚拟内存中，**所有内存以页为单位划分并且进行管理**，地址转换也以页为单位进行。

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，x86_64 架构中，页面大小为 4KB，这里假设一个页面大小



为 100 B(意味着之后的示意图，内存以每个 100 B 划分为一个格子)。

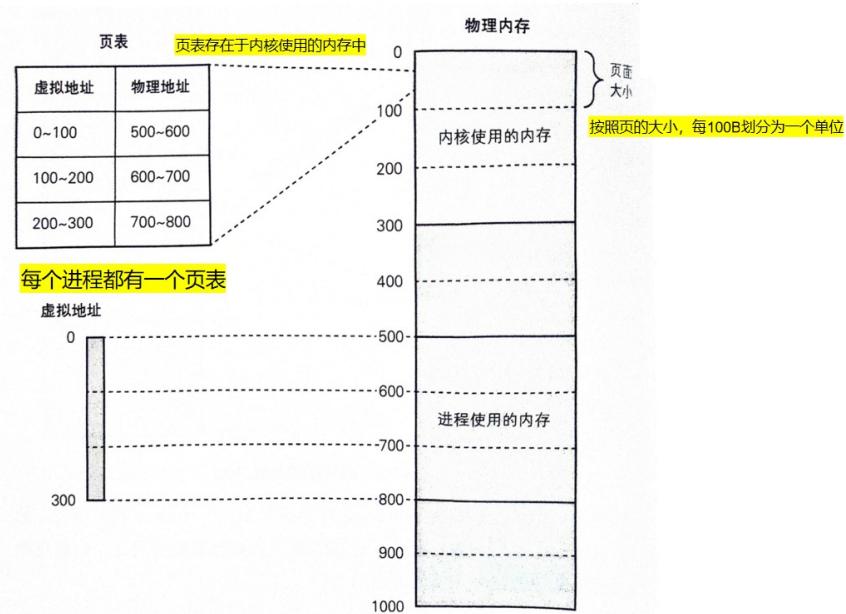


图 5-13 通过页表将虚拟地址映射到物理地址

如果该进程需要访问 0 ~ 300 的虚拟地址，CPU 会根据该进程的页表自动将其转换为相应的物理地址，**无需经过内核的处理**（页表是硬件实现的，以及用于加速的 TLB，软件实现的可能有）。

Q. 那么基于硬件的模块(实际上这个模块叫， MMU， 内存管理单元)，如何在不需要内核的下，甄别出不同的进程，从而切换到相应的页表来实现虚拟地址到实际地址的转换？

A. MMU 的工作流程为：

- **进程切换**：当操作系统进行进程切换时，会更新硬件中的一些寄存器，如页表基址寄存器 (Page Table Base Register, PTBR) 或者页表长度寄存器 (Page Table Length Register, PTLR)，以指向新进程的页表。
- **地址转换**：当一个进程访问内存时，CPU生成的虚拟地址会被MMU处理。MMU会使用当前进程的页表基址 (或其他相关信息) 来查找对应的页表项，进而将虚

拟地址转换为物理地址。

- **页表基址**: 在硬件中, MMU会根据当前进程的页表基址 (通常存储在PTBR中) 来定位到该进程的页表在内存中的位置。通过这个页表基址, MMU可以访问到正确的页表, 从而实现虚拟地址到物理地址的转换。
- **权限检查**: 在进行地址转换的过程中, MMU还会检查页表项中的权限位, 以确保进程对该内存区域有合适的访问权限。

如果, 上图进程强行访问大于 300 的虚拟地址, 则会发生 **地址越界** 问题, 如果虚拟地址空间为 500B, 形如:

虚拟地址	物理地址
0~100	500~600
100~200	600~700
200~300	700~800
300~400	x
400~500	x

此时 300 以上的区域还未分配物理内存, 此时访问大于 300 的虚拟地址, 会发生 **缺页中断**, 缺页中断会终止正在执行的命令, 启动内核中的缺页中断机构的处理。

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。

4.2 实验

编写一段访问非法地址的程序, 程序要求如下:

- 输出字符串 `before invalid access`
- 向必定会访问失败的地址 `NULL` 写入一个值 (这里将写入 0)
- 输出字符串 `after invalid access`

程序对于非法地址进行访问, 进而导致内核向进程发送 SIGSEGV 信号, 从而导致没

```

syz@syz:~/projects/class5$ cat segv.cpp
#include <iostream>

int main() {
    int *p = nullptr;
    puts("before invalid access");
    *p = 0;
    puts("after invalid access");
    return 0;
}
syz@syz:~/projects/class5$ ./segv
before invalid access
Segmentation fault
syz@syz:~/projects/class5$ █

```

有输出 `after invalid access` 程序就退出。

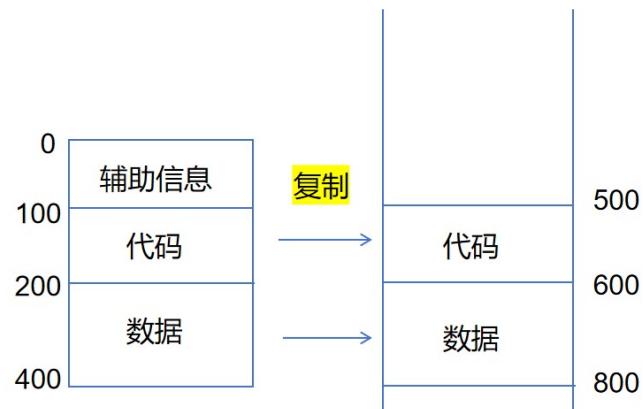
4.3 为进程分配内存

下面阐述，内核如何利用虚拟内存机制为进程分配内存。

1. 创建进程时

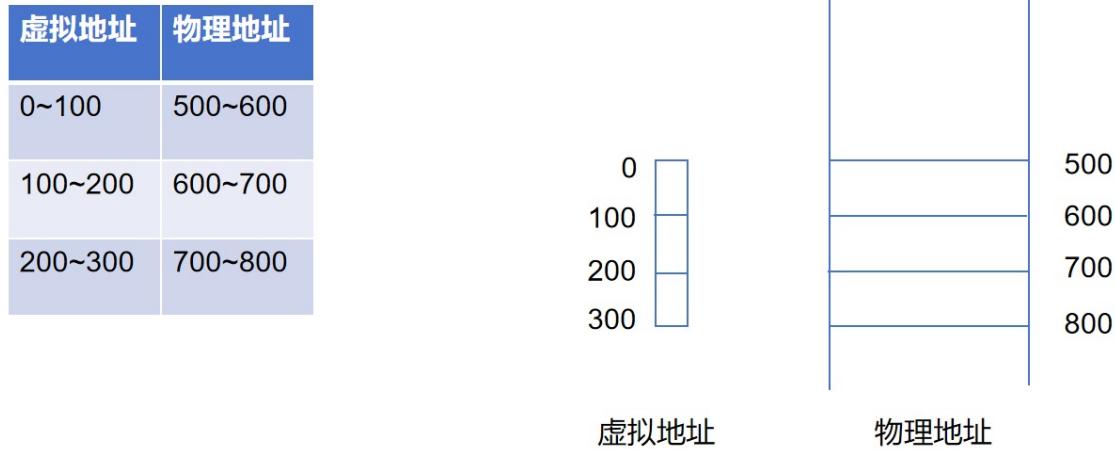
首先内核读取可执行文件，根据可执行文件的辅助信息将其 **代码段 + 数据段** 复制到内存上一块大小为 300 的区域，例如：

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



当复制完成后，根据当前代码段和数据段在内存的实际地址，创建该进程的页表，将虚拟地址映射到物理地址。

然后从程序入口处地址 **0**，对应实地址 **500** 开始运行程序。



2. 在动态分配内存

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，然后把新分配的内存的物理地址对应的虚拟地址返回给进程，例如：

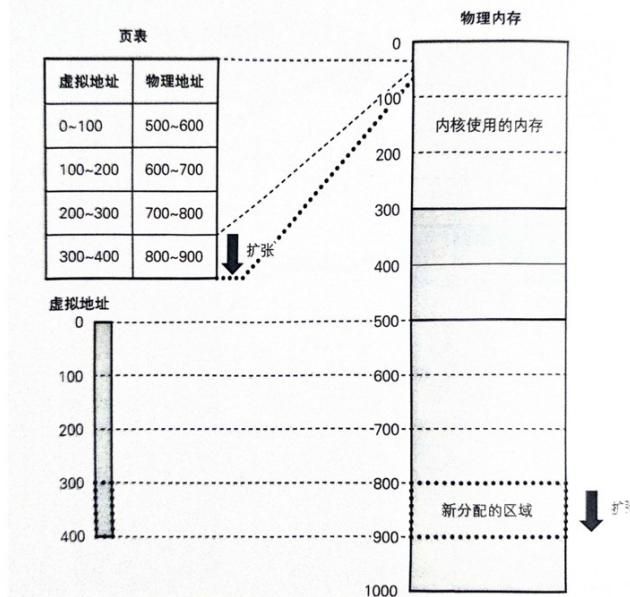


图 5-19 动态分配内存 (存在虚拟内存时的情形)

4.4 实验

我们需要实现以下 3 个功能的程序：

- 显示进程的内存映射信息：(`/proc/[pid]/maps` 的输出)
- 额外回获取 **100mb** 的内存
- 再次显示内存映射的信息

mmap.cpp

```
#include <unistd.h>
#include <sys/mman.h>
#include <bits/stdc++.h>
#include <err.h>

constexpr int BUFFER_SIZE = 1000;
constexpr int ALLOC_SIZE = 100 * 1024 * 1024;

static char command[BUFFER_SIZE];

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

    std::cout << "*** Mem map before allocation ***" << std::endl;
    system(command);

    void *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");
    std::cout << std::endl;

    printf("*** succeeded to allocate memory: address = %p; size = 0%x\n",
new_memory, ALLOC_SIZE);
    std::cout << std::endl;

    std::cout << "*** Mem map after allocation ***" << std::endl;
    system(command);

    // 解除申请的 100mb 的内存
    if (munmap(new_memory, ALLOC_SIZE) == -1)
        err(EXIT_FAILURE, "munmap() failed");
    return 0;
}
```

`system()` 实现了一种简单的类似 fork-and-exec 的功能

运行编译后的可执行程序，得到：

```
*** Mem map before allocation ***  
...  
55c818a13000-55c818a34000 rw-p 00000000 00:00 0  
[heap]  
7f3d6c9fd000-7f3d6ca01000 rw-p 00000000 00:00 0  
...  
  
*** succeeded to allocate memory: address = 0x7f3d665fd000; size = 0x6400000  
***  
  
*** Mem map after allocation ***  
...  
55c818a13000-55c818a34000 rw-p 00000000 00:00 0  
[heap]  
7f3d665fd000-7f3d6ca01000 rw-p 00000000 00:00 0  
....
```

可以发现申请 `100MB` 空间后，`7f3d6c9fd000-7f3d6ca01000` 变成了 `7f3d665fd000-7f3d6ca01000`，多出了内存：

然后通过

```
syz@syz:~/projects/class5$ python3 -c "print(- 0x7f3d665fd000 +  
0x7f3d6c9fd000)"  
104857600
```

4.5 利用上层进行分配

c 语言中存在一个名为 `malloc` 的函数，也是内存申请相关，Linux 中，该函数底层用到了 `mmap()`

`mmap()` 是以页为单位获取内存的，而 `malloc()` 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()` 向内核申请一大块内存区域作为内存池，程序调用 `malloc`，`glibc` 会划分相应内存给程序，当内存池内存消耗完，`glibc` 会再次调用 `mmap()` 申请新的内存区域。

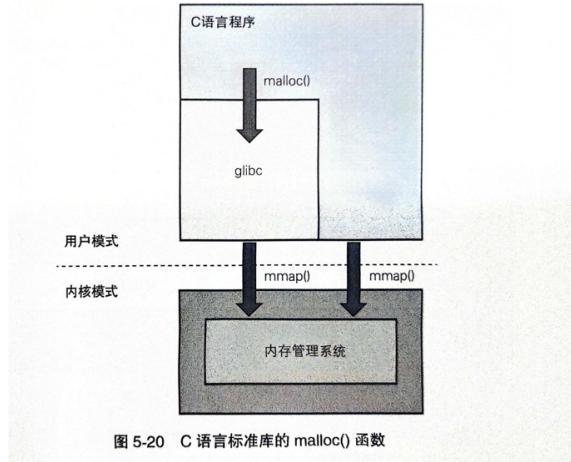


图 5-20 C 语言标准库的 `malloc()` 函数

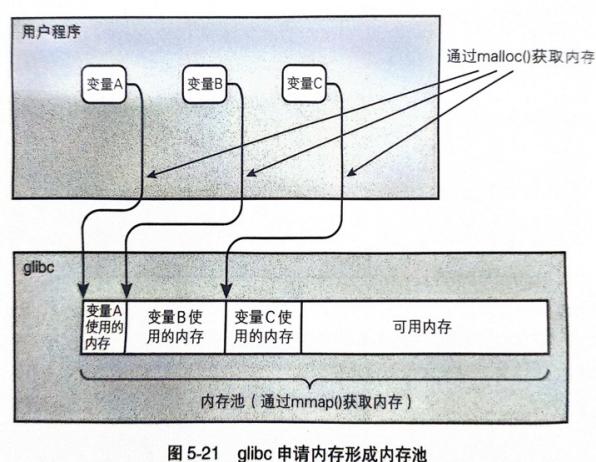


图 5-21 `glibc` 申请内存形成内存池

一般来说 Linux 显示进程的内存消耗大于程序统计自身内存消耗量，程序自身内存消耗量一般统计 `malloc()` 申请的那部分，而 Linux 显示的是创建进程消耗的内存，以及进程通过 `mmap()` 分配的所有内存。

即使是 python 其底层，仍然是通过 c 语言的 `malloc` 来获取内存。

4.6 解决问题

内存碎片化

只需要设计好页表，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，类似：

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，页表也是每个进程独有的，因此进程根本无法访问其它进程的内存，类似：

但是一般处于方便，内核的内存区域被映射到了所有进程的虚拟地址空间中。但是与内核相关的页表项都标有“内核模式专用”的信息，只有在 CPU 处于内核模式才能进行访问，因此这部分内存也不会在用户模式下被意外访问，类似：

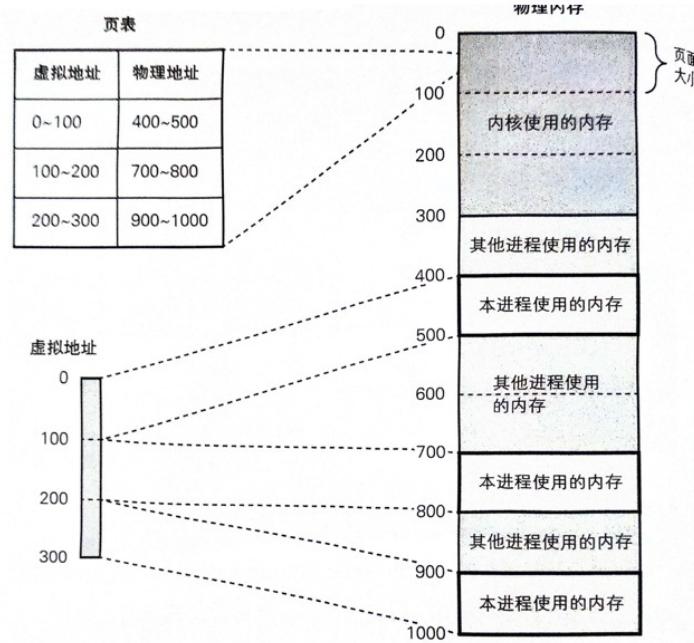


图 5-23 通过虚拟内存解决内存碎片化问题

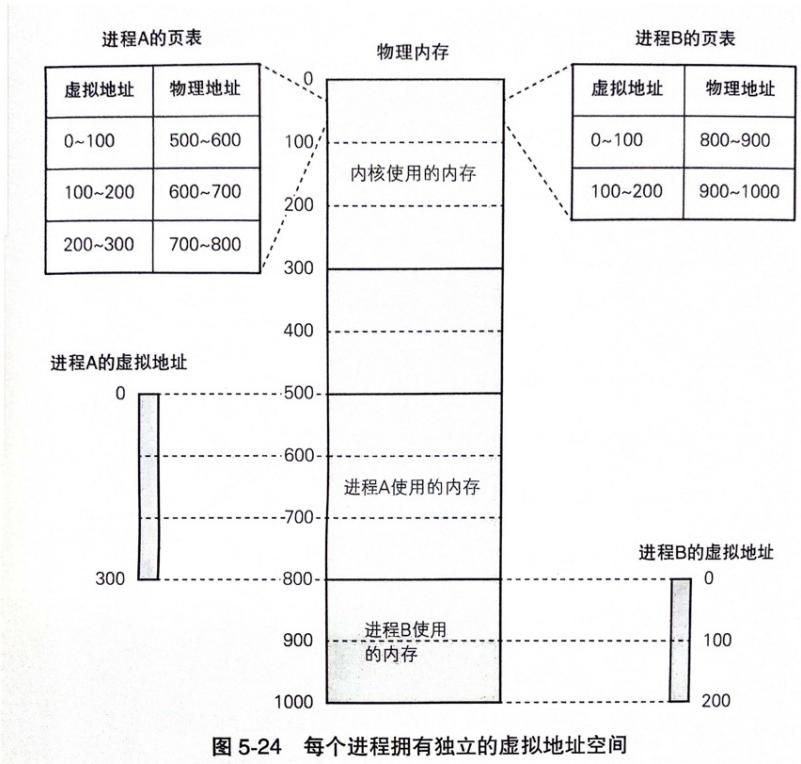


图 5-24 每个进程拥有独立的虚拟地址空间

难以执行多任务

每个进程拥有独立的虚拟地址空间，不用担心干扰其它程序的运行。

5. 虚拟内存的应用

虚拟内存有 6 大重要功能：

- 文件映射
- 请求分页

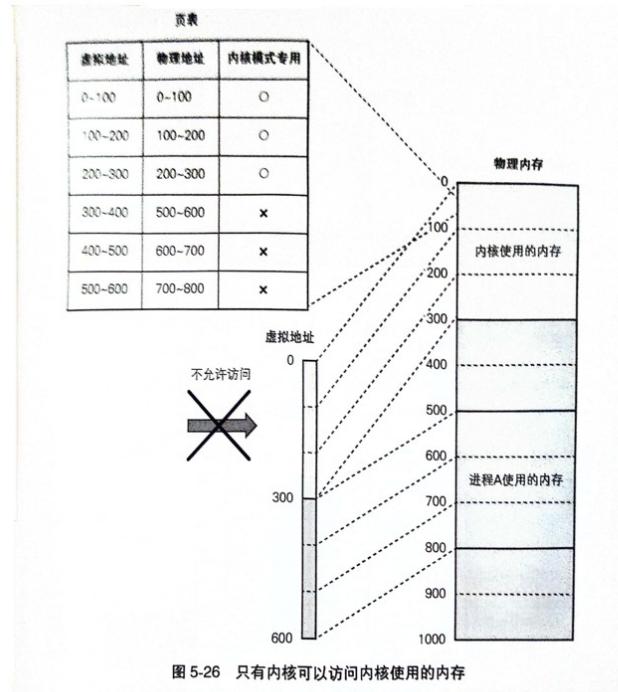


图 5-26 只有内核可以访问内核使用的内存

- 利用写时复制快速创建进程
- Swap
- 多级页表
- 标准大页

5.1 文件映射

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

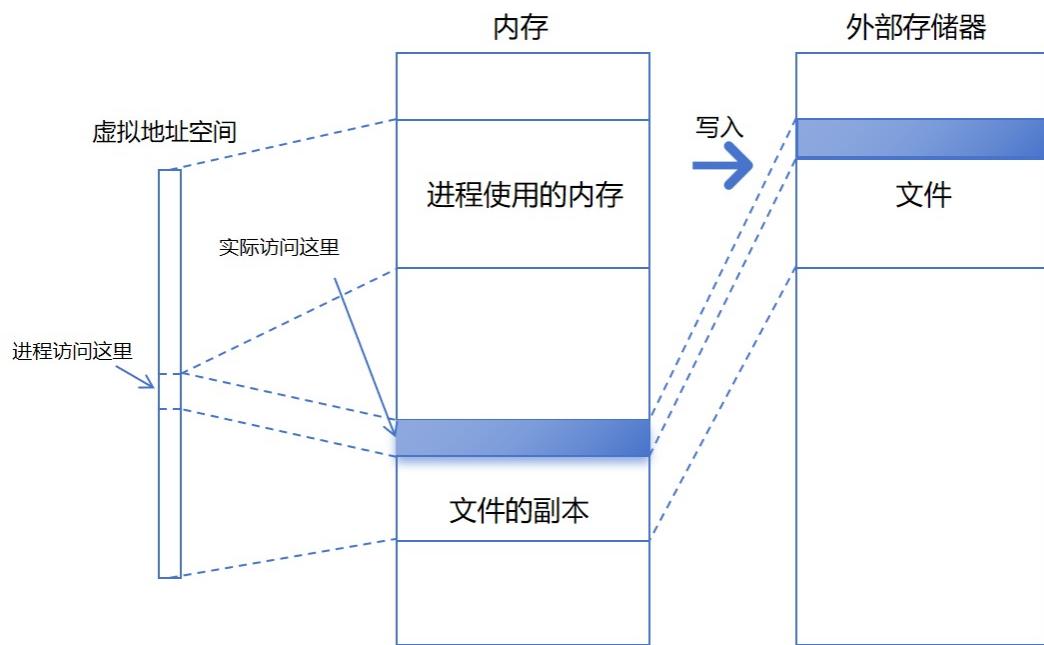
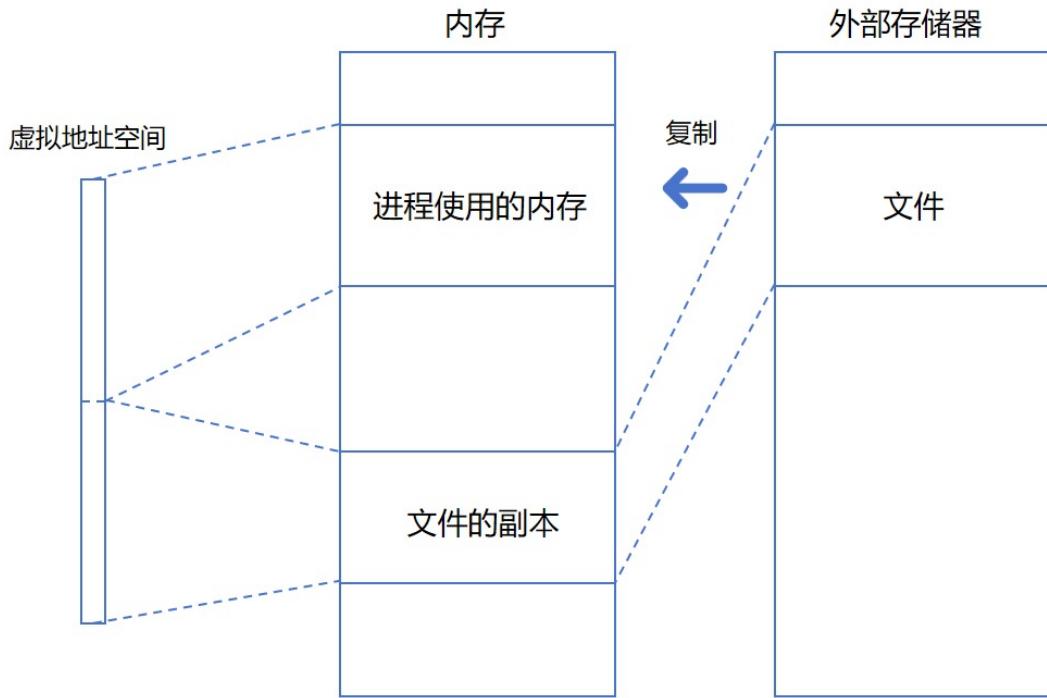
首先，按照指定方式调用 `mmap()` 函数，即可将文件的内容读取到内存中，然后把这个内存区域映射到虚拟地址空间：

接着，在进程内可以通过访问内存的方式来访问被映射的文件，被访问的区域会在规定的时间写入外部存储器上的文件。

实验

编写一个使用文件映射功能的程序，确认是否真的能映射文件，以及是否能成功访问文件内容，需要确认：

- 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
- 能否通过向映射的区域写入数据来将数据写入文件



首先介绍认真介绍一下 **mmap()** 函数：

mmap 函数用于将一个文件或者其它对象映射到内存中。其函数签名如下：

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

参数说明如下：

- **addr** : 指定映射区域的开始地址，通常设置为 0，表示由系统选择合适的地址。
- **length** : 指定映射区域的长度，单位是字节。
- **prot** : 指定映射区域的保护方式，可以是以下值的按位或组合：
 - **PROT_READ** : 映射区域可被读取
 - **PROT_WRITE** : 映射区域可被写入
 - **PROT_EXEC** : 映射区域可被执行
 - **PROT_NONE** : 映射区域不能被访问
- **flags** : 指定映射区域的标志，可以是以下值的按位或组合：
 - **MAP_SHARED** : 对映射区域的写入会更新文件内容，并且可以被其它映射该文件的进程所共享
 - **MAP_PRIVATE** : 对映射区域的写入不会更新文件内容，并且不会被其它映射该文件的进程所共享
 - **MAP_ANONYMOUS** : 匿名映射，映射的是内存而不是文件
- **fd** : 指定要映射的文件描述符。如果是匿名映射，则设置为 -1。
- **offset** : 指定文件映射的偏移量，通常设置为 0 表示从文件的起始位置开始映射。

filemap.cpp

```
#include <unistd.h>
#include <sys/mman.h>
#include <err.h>
#include <iostream>
#include <fcntl.h> // open() close() 函数归属处
#include <cstring>

constexpr int BUFFER_SIZE = 1000;
constexpr int ALLOC_SIZE = 100 * 1024 * 1024;

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

int main() {
```

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

std::cout << "Mem map before mapping file" << std::endl;
system(command);

int fd = open("testfile", O_RDWR);
if (fd == -1)
    err(EXIT_FAILURE, "open() failed");
// 这里必须强转
char *file_contents = (char*)mmap(NULL, ALLOC_SIZE, PROT_READ |
PROT_WRITE, MAP_SHARED, fd, 0);
if (file_contents == (void *) -1) {
    warn("mmap() failed");
    if (close(fd) == -1)
        warn("close() failed");
    return 0;
}

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

std::cout << "*** Mem map after mapping ***" << std::endl;
system(command);
std::cout << std::endl;

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

// 改写映射文件
memcpy(file_contents, overwrite_data, strlen(overwrite_data));
std::cout << std::endl;

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

if (munmap(file_contents, ALLOC_SIZE) == -1)
    warn("munmap() failed");
if (close(fd) == -1)
```

```
        warn("close() failed");
    return 0;
}
```

编译后运行得到结果：

```
Mem map before mapping file
...
559777f18000-559777f39000 rw-p 00000000 00:00 0
[heap]
7fc991ffb000-7fc991fff000 rw-p 00000000 00:00 0
...
*** succeeded to map file : address = 0x7fc98bbfb000; size = 0x6400000 ***
...
559777f18000-559777f39000 rw-p 00000000 00:00 0
[heap]
7fc98bbfb000-7fc991ffb000 rw-s 00000000 08:20 2367
/home/syz/projects/class5/testfile
7fc991ffb000-7fc991fff000 rw-p 00000000 00:00 0
...
```

`python3 -c "print(0x7fc991ffb000 - 0x7fc98bbfb000)"` 结果为 `104857600`，显示内核给该文件分配了 `100mb` 内存。

`cat testfile` 结果显示 `HELLO`

也可以使用 `write()`, `fprintf()` 函数进行写

5.2 请求分页

之前，我们提及创建进程的内存分配，以及创建进程后通过 `mmap()` 系统调用进行的动态内存分配，是这样描述的：

- 内核直接从物理内存中获取需要的区域

- 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，获取的内存中，有一部分甚至直到进程运行结束都不会被使用，例如：

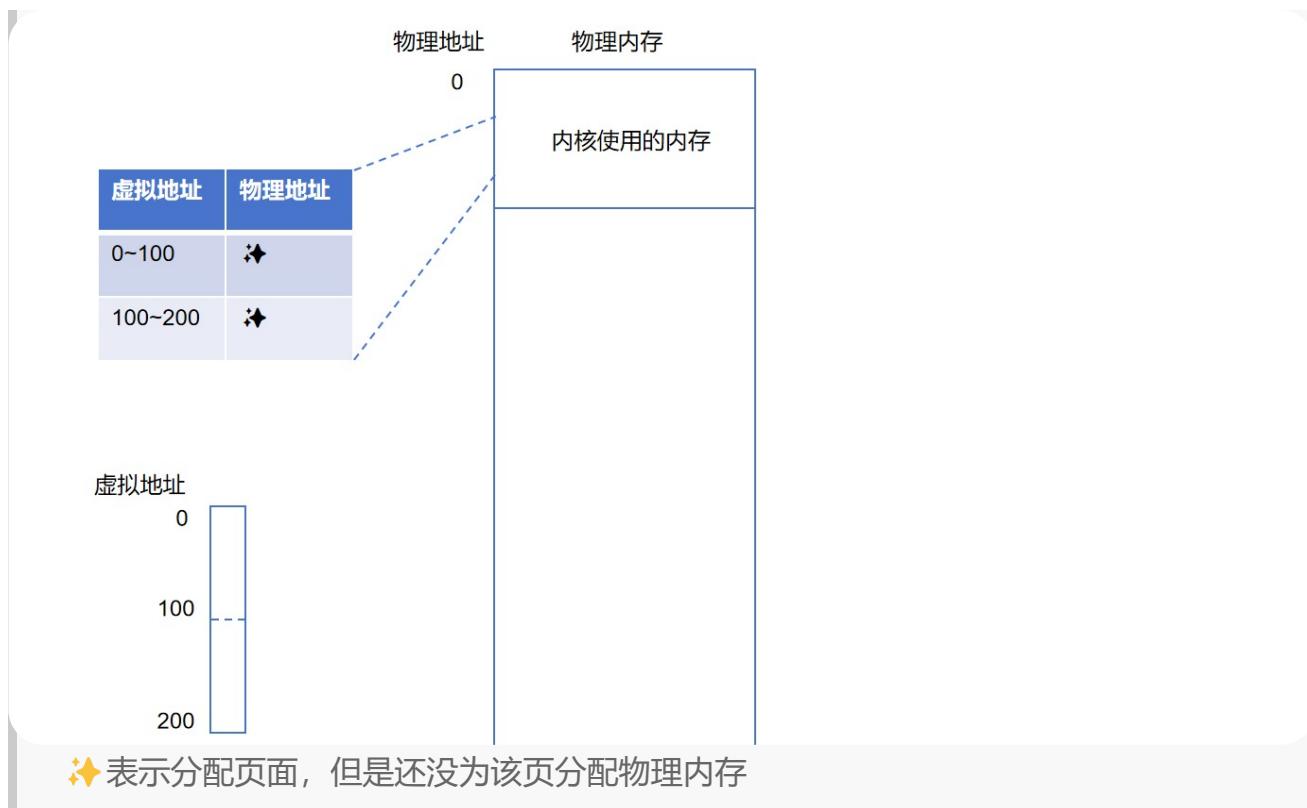
- 用于大规模程序中，程序运行时未使用的功能的代码段和数据段
- 由 **glibc** 保留的内存池中未被用户利用的部分

Linux 采用 **请求分页** 来为进程分配内存

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。页面状态除了前面提到的“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。

下面具体描述这个流程：

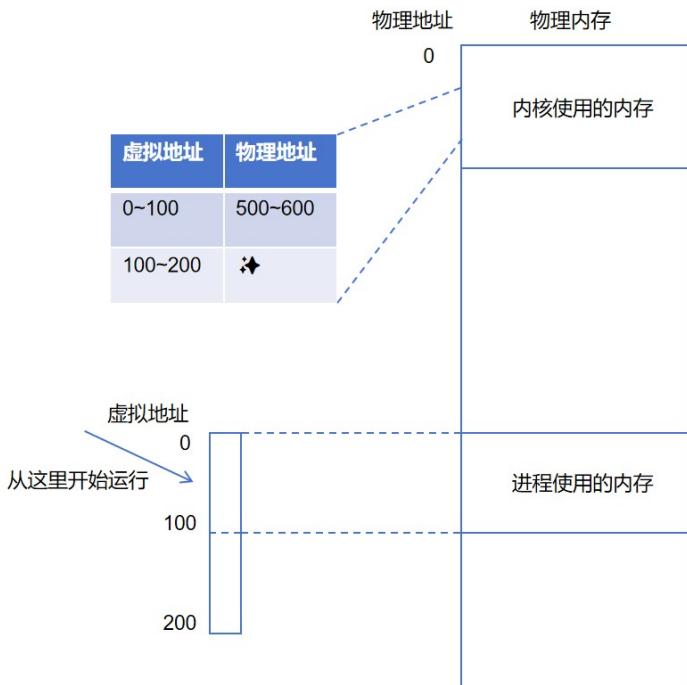
在创建进程时，其虚拟地址空间中的与代码段和数据段对应的页面上，添加“**已为进程分配该区域(页面)**”这样的信息，但暂时不会为其分配物理内存，例如：



当程序从入口处开始运行进程，为入口处所属页面分配物理内存：

此时处理流程为：

- 进程访问入口点



- CPU 参照页表，筛选出 入口点 所属页面中哪些虚拟地址未关联物理地址
- 在 CPU 中引发缺页中断
- 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。
- 回到用户模式，继续运行进程

此外，进程本身无法感知自身在运行时曾发生缺页中断

此后，每当访问新的区域，都按照上述流程，先触发缺页中断，然后分配物理内存，更新对应页表。

当进程动态申请新的内存时：

首先通过 `mmap()` 函数动态获取内存，但此时只是获取到虚拟地址，还未分配物理内存：

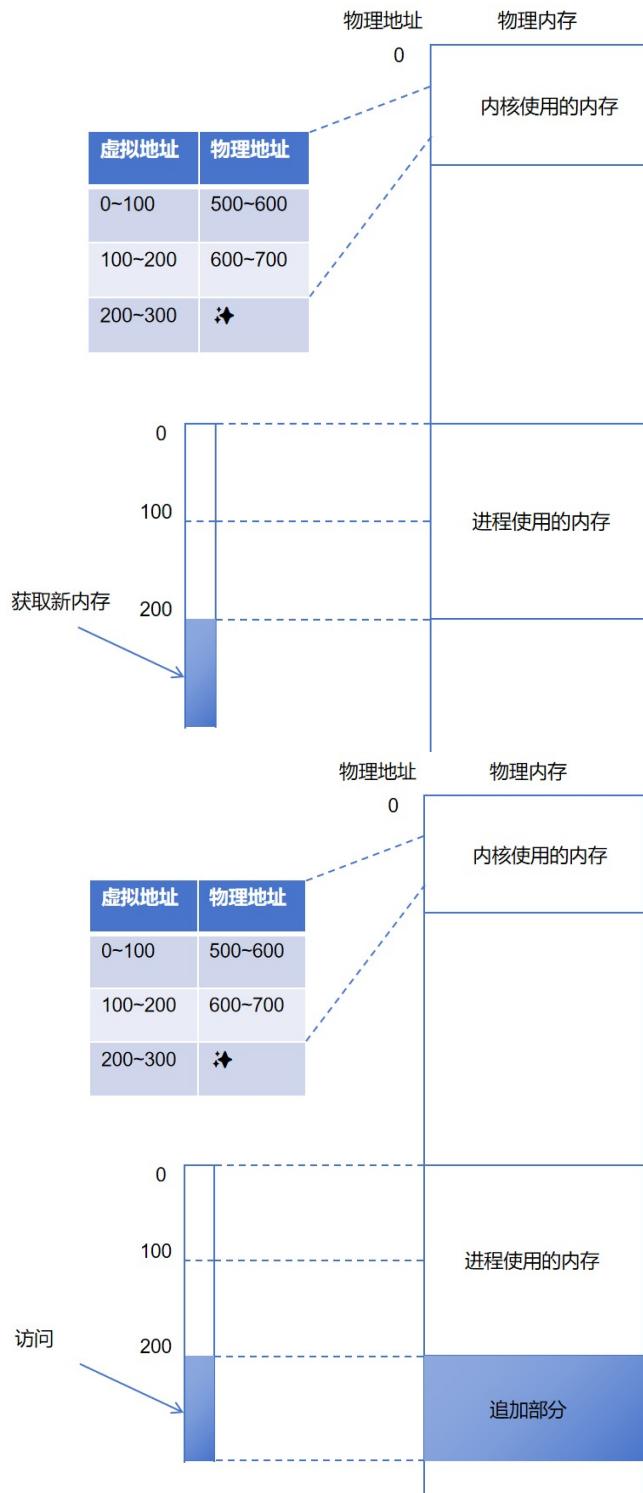
然后，如果对该内存发起访问，进程就会为其分配物理内存：

总结：

- 进程通过 `mmap()` 函数等成功获取内存，其实是成功获取虚拟内存
- 访问所获取的虚拟内存，并将虚拟内存关联到物理内存，其实是成功获取物理内存
- 即使 `mmap()` 函数调用成功，在向内存写数据时，如果物理内存没有充足的可用内存就会引发物理内存的问题

实验

我们需要确认以下事实：



- 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？

可以考虑编写实现如下要求的程序：

- 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
- 获取 100mb 内存
- 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

- 从头到尾逐页访问已获取的内存，每访问 10mb 内存，就输出一条信息，用于提示当前访问进度
- 在访问完所有获取的内存后，输出相应的提示信息，等待用户按下 enter 键，在每条信息的开头添加时间戳。

demand-paging.cpp

```
#include <unistd.h>
#include <bits/stdc++.h>
#include <err.h>
#include <time.h>

constexpr int BUFFER_SIZE = 100 * 1024 * 1024;
constexpr int NCYCLE = 10;
constexpr int PAGE_SIZE = 4096;

int main() {
    time_t t = time(NULL); char *s = ctime(&t);
    // 避免输出换行
    printf("[%.*s]: before allocation, please press Enter key\n", (int)
(strlen(s) - 1), s);
    getchar();

    char *p = (char*) malloc(BUFFER_SIZE); // malloc 按照字节分配内存
    if (!p) err(EXIT_FAILURE, "malloc failed");

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

    // char 一个位置一个字节
    for (int i = 0; i < BUFFER_SIZE; i++) {
        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]: page %d, at time %s\n", (int)(strlen(s) - 1), s, cycle, ctime(&t));
        }
    }
}
```

```

        printf("[%.*s]: touched %dMB\n", (int)(strlen(s) -
1), s, i / (1024 * 1024));
        sleep(1); // 休眠 1s
    }

    t = time(NULL), s = ctime(&t);
    printf("[%.*s]: touched %dMB, please press ENTER key\n", (int)
(strlen(s) - 1), s,\n
        BUFFER_SIZE / (1024 * 1024));
    getchar(), free(p);
    return 0;
}

}

```

单独这一个程序并不能实现我们的目标，我们需要同时在另一个 `bash` 上使用 `sar -r 1` (开两个终端即可实现)，每隔 1s 输出内存相关信息：

```

// bash1 的内容
syz@syz:~/projects/class5$ ./demand-paging
[Thu Feb 29 20:38:38 2024]: before allocation, please press Enter key

[Thu Feb 29 20:38:39 2024]: allocated 100MB, please Enter key

[Thu Feb 29 20:38:43 2024]: touched 10MB
[Thu Feb 29 20:38:44 2024]: touched 20MB
[Thu Feb 29 20:38:45 2024]: touched 30MB
[Thu Feb 29 20:38:46 2024]: touched 40MB
[Thu Feb 29 20:38:47 2024]: touched 50MB
[Thu Feb 29 20:38:48 2024]: touched 60MB
[Thu Feb 29 20:38:49 2024]: touched 70MB
[Thu Feb 29 20:38:50 2024]: touched 80MB
[Thu Feb 29 20:38:51 2024]: touched 90MB
[Thu Feb 29 20:38:52 2024]: touched 100MB, please press ENTER key

```

```

// bash2 的内容
syz@syz:~$ sar -r 1
Linux 5.15.133.1-microsoft-standard-WSL2 (syz) 02/29/24      _x86_64_
(20 CPU)

```

	kbmemfree	kbavail	kbmemused	%memused	kbbuffers	kbcached
20:38:30						
kbccommit	%commit	kbactive	kbinact	kbdirty		
20:38:31	6987152	7169280	468140	5.89	17284	366340
825936	8.23	155716	426892	0		
20:38:32	6985124	7167252	470168	5.92	17284	366340
825936	8.23	155716	426892	0		
20:38:33	6985124	7167304	470168	5.92	17284	366340
825936	8.23	155768	426892	0		
20:38:34	6985124	7167304	470168	5.92	17284	366340
825936	8.23	155768	426892	0		
20:38:35	6985124	7167304	470168	5.92	17284	366340
825936	8.23	155768	426892	0		
20:38:36	6985124	7167304	470168	5.92	17284	366340
825936	8.23	155768	426892	0		
20:38:37	6985124	7167304	470168	5.92	17284	366340
825936	8.23	155768	426892	0		
20:38:38	6985124	7167304	470168	5.92	17284	366340
826408	8.23	155768	426892	0 // <- 进入程序		
20:38:39	6985124	7167304	470168	5.92	17284	366340
826408	8.23	155772	427040	0		
20:38:40	6985124	7167304	470168	5.92	17284	366340
928812	9.25	155772	427040	0		
20:38:41	6985124	7167304	470168	5.92	17284	366340
928812	9.25	155772	427040	0		
20:38:42	6985124	7167304	470168	5.92	17284	366340
928812	9.25	155772	427040	0		
20:38:43	6974904	7157084	480388	6.05	17292	366332
928812	9.25	155772	437716	0 // <- 开始访问申请内存		
20:38:44	6966712	7148896	488572	6.15	17292	366340
928812	9.25	155776	447980	4		
20:38:45	6954424	7136608	500860	6.31	17292	366340
928812	9.25	155776	458220	4		
20:38:46	6946232	7128416	509052	6.41	17292	366340
928812	9.25	155776	468460	4		
20:38:47	6933944	7116128	521340	6.56	17292	366340
928812	9.25	155776	478700	4		
20:38:48	6925752	7107936	529532	6.67	17292	366340
928812	9.25	155776	488940	0		

20:38:49	6913464	7095648	541820	6.82	17292	366340
928812	9.25	155776	499180	0		
20:38:50	6905272	7087456	550012	6.92	17292	366340
928812	9.25	155776	509420	0		
20:38:51	6892984	7075168	562300	7.08	17292	366340
928812	9.25	155776	519660	0		
20:38:52	6883532	7065716	571752	7.20	17292	366340
928812	9.25	155776	529316	0		
20:38:53	6881504	7063688	573780	7.22	17292	366340
928812	9.25	155776	529500	0 // <- free 然后退出进程		
20:38:54	6985912	7168100	469372	5.91	17292	366340
825936	8.23	155776	426892	0		
20:38:55	6985912	7168100	469372	5.91	17292	366340
825936	8.23	155776	426892	0		
20:38:56	6985912	7168100	469372	5.91	17292	366340
825936	8.23	155776	426892	0		

得出结论：

- 即使已经获取内存区域，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) 也几乎不会发生改变。
- 在开始访问内存后，内存使用量每秒增加 `10mb` 左右
- 在访问结束后，内存使用量不再发生变化。
- 在进程运行结束后，内存使用量回到开始运行进程的状态(可能稍微有些区别)

接下来使用同样的做法，一个终端上运行程序，另一个终端上使用 `sar -B` 命令每秒观测一次缺页中断的发生情况。

```
// [bash1]:
syz@syz:~/projects/class5$ ./demand-paging
[Thu Feb 29 20:47:26 2024]: before allocation, please press Enter key

[Thu Feb 29 20:47:27 2024]: allocated 100MB, please Enter key

[Thu Feb 29 20:47:29 2024]: touched 10MB
[Thu Feb 29 20:47:30 2024]: touched 20MB
[Thu Feb 29 20:47:31 2024]: touched 30MB
[Thu Feb 29 20:47:32 2024]: touched 40MB
```

```
[Thu Feb 29 20:47:33 2024]: touched 50MB
[Thu Feb 29 20:47:34 2024]: touched 60MB
[Thu Feb 29 20:47:35 2024]: touched 70MB
[Thu Feb 29 20:47:36 2024]: touched 80MB
[Thu Feb 29 20:47:37 2024]: touched 90MB
[Thu Feb 29 20:47:38 2024]: touched 100MB, please press ENTER key
```

// [bash2]:

```
syz@syz:~$ sar -B 1
Linux 5.15.133.1-microsoft-standard-WSL2 (syz) 02/29/24           _x86_64_
(20 CPU)
```

	pgpgin/s	pgpgout/s	fault/s	majflt/s	pgfree/s	pgscank/s	pgscand/s	pgsteal/s	%vmeff
20:47:14									
20:47:15	0.00	0.00	0.00	0.00	0.00	30.69			0.00
	0.00	0.00	0.00						
20:47:16	0.00	0.00	0.00	1.00	0.00	0.00			0.00
	0.00	0.00	0.00						
20:47:17	0.00	0.00	0.00	1.00	0.00	0.00			0.00
	0.00	0.00	0.00						
20:47:18	0.00	0.00	0.00	203.00	0.00	124.00			0.00
	0.00	0.00	0.00						
// <- 开始访问内存									
20:47:19	0.00	0.00	0.00	0.00	0.00	0.00			0.00
	0.00	0.00	0.00						
20:47:20	0.00	0.00	0.00	1.00	0.00	0.00			0.00
	0.00	0.00	0.00						
20:47:21	0.00	0.00	0.00	9.00	0.00	0.00			0.00
	0.00	0.00	0.00						
20:47:22	0.00	0.00	0.00	0.00	0.00	0.00			0.00
	0.00	0.00	0.00						
20:47:23	0.00	0.00	0.00	0.00	0.00	0.00			0.00
	0.00	0.00	0.00						
20:47:24	0.00	0.00	0.00	0.00	0.00	0.00			0.00
	0.00	0.00	0.00						
20:47:25	0.00	0.00	0.00	1.00	0.00	1.00			0.00
	0.00	0.00	0.00						
20:47:26	0.00	0.00	0.00	183.00	0.00	50.00			0.00
	0.00	0.00	0.00						

20:47:27	0.00	0.00	1.00	0.00	0.00	0.00
0.00	0.00	0.00				
20:47:28	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00				
20:47:29	0.00	0.00	434.00	0.00	0.00	0.00
0.00	0.00	0.00				
20:47:30	0.00	0.00	5.00	0.00	0.00	0.00
0.00	0.00	0.00				
20:47:31	0.00	0.00	5.00	0.00	278.00	0.00
0.00	0.00	0.00				
20:47:32	0.00	0.00	5.00	0.00	0.00	0.00
0.00	0.00	0.00				
20:47:33	0.00	0.00	5.00	0.00	0.00	0.00
0.00	0.00	0.00				
20:47:34	0.00	0.00	5.00	0.00	0.00	0.00
0.00	0.00	0.00				
20:47:35	0.00	0.00	5.00	0.00	1.00	0.00
0.00	0.00	0.00				
20:47:36	0.00	0.00	5.00	0.00	0.00	0.00
0.00	0.00	0.00				
20:47:37	0.00	0.00	5.00	0.00	0.00	0.00
0.00	0.00	0.00				
20:47:38	0.00	0.00	88.00	0.00	0.00	0.00
0.00	0.00	0.00				
// <- 结束						
20:47:39	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00				
20:47:40	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00				

可以看到，在进程访问所获取的内存区域这段时间，每秒发生的中断次数 (`fault/s`) 字段的值有所增加；另外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(但是本实验没有体现这一点)。

接下来，我们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：

- 虚拟内存量，可通过 `ps -eo vsz` 获取

- 已分配的物理内存量，`ps -eo rss`
- 以及在创建进程后发生缺页中断的总次数，可通过`ps -eo maj_flt,min_flt`获取

```
vsz-rss.sh
```

```
#!/bin/bash

# 第一行告诉操作系统使用 bash 来执行该脚本
while true ; do
    # 将当前日期字符串去掉换行符后，存在变量 DATE
    DATE=$(date | tr -d '\n')
    # 获得 ps -eo xxx 中含 demand-paging 的一整行的结果，grep -v 作用暂且不知
    INFO=$(ps -eo pid,comm,vsz,rss,maj_flt,min_flt | grep demand-paging
    | grep -v grep)
    # 条件控制语句，如果 "INFO" 字段为空，则退出
    if [ -z "$INFO" ] ; then
        echo "$DATE: target process seems to be finished"
        break
    fi
    echo "${DATE}: ${INFO}"
    # 睡眠 1s 让人看起来舒服
    sleep 1
done
```

接着使用`chmod +x vsz-rss.sh`增加可执行的权限，在运行`demand-paging`之后，先不要开始申请虚拟内存，然后运行`./vsz-rss.sh`，观察输出：

```
[bash1]
syz@syz:~/projects/class5$ ./demand-paging
[Thu Feb 29 21:31:13 2024]: before allocation, please press Enter key

[Thu Feb 29 21:36:08 2024]: allocated 100MB, please Enter key

[Thu Feb 29 21:36:12 2024]: touched 10MB
[Thu Feb 29 21:36:13 2024]: touched 20MB
[Thu Feb 29 21:36:14 2024]: touched 30MB
```

```
[Thu Feb 29 21:36:15 2024]: touched 40MB
[Thu Feb 29 21:36:16 2024]: touched 50MB
[Thu Feb 29 21:36:17 2024]: touched 60MB
[Thu Feb 29 21:36:18 2024]: touched 70MB
[Thu Feb 29 21:36:19 2024]: touched 80MB
[Thu Feb 29 21:36:20 2024]: touched 90MB
[Thu Feb 29 21:36:21 2024]: touched 100MB, please press ENTER key
```

[bash2]

```
syz@syz:~/projects/class5$ ./vsz-rss.sh
# 时间 & pid & 命令 & vsz(虚拟内存量) & rss(已分配的物理内存量) & maj_flt(硬性页缺失次数) & min_flt(软性页缺失次数)

Thu Feb 29 21:35:55 CST 2024:      939 demand-paging      6056  2064      0
    152
Thu Feb 29 21:35:56 CST 2024:      939 demand-paging      6056  2064      0
    152
Thu Feb 29 21:35:57 CST 2024:      939 demand-paging      6056  2064      0
    152
Thu Feb 29 21:35:58 CST 2024:      939 demand-paging      6056  2064      0
    152
Thu Feb 29 21:35:59 CST 2024:      939 demand-paging      6056  2064      0
    152
Thu Feb 29 21:36:00 CST 2024:      939 demand-paging      6056  2064      0
    152
Thu Feb 29 21:36:01 CST 2024:      939 demand-paging      6056  2064      0
    152
Thu Feb 29 21:36:02 CST 2024:      939 demand-paging      6056  2064      0
    152
Thu Feb 29 21:36:03 CST 2024:      939 demand-paging      6056  2064      0
    152
Thu Feb 29 21:36:04 CST 2024:      939 demand-paging      6056  2064      0
    152
Thu Feb 29 21:36:05 CST 2024:      939 demand-paging      6056  2064      0
    152
Thu Feb 29 21:36:06 CST 2024:      939 demand-paging      6056  2064      0
    152
Thu Feb 29 21:36:07 CST 2024:      939 demand-paging      6056  2064      0
    152
Thu Feb 29 21:36:08 CST 2024:      939 demand-paging     108460  2064      0
    153 # <- 开始分配虚拟内存
```

Thu Feb 29 21:36:09 CST 2024:	939	demand-paging	108460	2064	0
153					
Thu Feb 29 21:36:10 CST 2024:	939	demand-paging	108460	2064	0
153					
Thu Feb 29 21:36:11 CST 2024:	939	demand-paging	108460	2064	0
153					
Thu Feb 29 21:36:12 CST 2024:	939	demand-paging	108460	14872	0
499 # <- 开始逐渐分配物理内存					
Thu Feb 29 21:36:13 CST 2024:	939	demand-paging	108460	25112	0
504					
Thu Feb 29 21:36:14 CST 2024:	939	demand-paging	108460	35352	0
509					
Thu Feb 29 21:36:15 CST 2024:	939	demand-paging	108460	45856	0
514					
Thu Feb 29 21:36:16 CST 2024:	939	demand-paging	108460	56096	0
519					
Thu Feb 29 21:36:17 CST 2024:	939	demand-paging	108460	66336	0
524					
Thu Feb 29 21:36:18 CST 2024:	939	demand-paging	108460	76576	0
529					
Thu Feb 29 21:36:19 CST 2024:	939	demand-paging	108460	86816	0
534					
Thu Feb 29 21:36:20 CST 2024:	939	demand-paging	108460	97056	0
539					
Thu Feb 29 21:36:21 CST 2024:	939	demand-paging	108460	105904	0
715 # <- 100MB物理内存分配完毕					
Thu Feb 29 21:36:22 CST 2024:	939	demand-paging	108460	105904	0
715					
Thu Feb 29 21:36:23 CST 2024:	939	demand-paging	108460	105904	0
715					
Thu Feb 29 21:36:24 CST 2024:	939	demand-paging	108460	105904	0
715					
Thu Feb 29 21:36:25 CST 2024:	939	demand-paging	108460	105904	0
715					
Thu Feb 29 21:36:26 CST 2024:	939	demand-paging	108460	105904	0
715					
Thu Feb 29 21:36:27 CST 2024:	939	demand-paging	108460	105904	0
715					
Thu Feb 29 21:36:28 CST 2024:	939	demand-paging	108460	105904	0
715					

```
Thu Feb 29 21:36:29 CST 2024:      939 demand-paging    108460 105904    0
715
Thu Feb 29 21:36:30 CST 2024:      939 demand-paging    108460 105904    0
715
Thu Feb 29 21:36:31 CST 2024:      939 demand-paging    108460 105904    0
715
Thu Feb 29 21:36:32 CST 2024:      939 demand-paging    108460 105904    0
715
Thu Feb 29 21:36:33 CST 2024: target process seems to be finished
```

综上，得到以下结论：

- 在已经获取了内存，但是尚未进行访问这段时间内，虚拟内存量比获取前增加了 **100MB**，物理内存量并没有变化
- 在开始访问内存后，物理内存量每秒增加 **10MB** 左右，但虚拟内存量没有发生变化
- 访问结束时候，物理内存量比访问前多了 **100MB**

虚拟内存不足与物理内存不足

一般，进程在运行中，如果获取内存失败，进程就会异常终止，获取内存失败也分为虚拟内存不足以及物理内存不足。

比如：当虚拟地址空间大小只有 **500B**，并且虚拟内存已经被耗尽(被申请完)，即使还有物理内存剩余，也会引发虚拟内存不足。

虚拟内存不足与剩余多少物理内存无关。

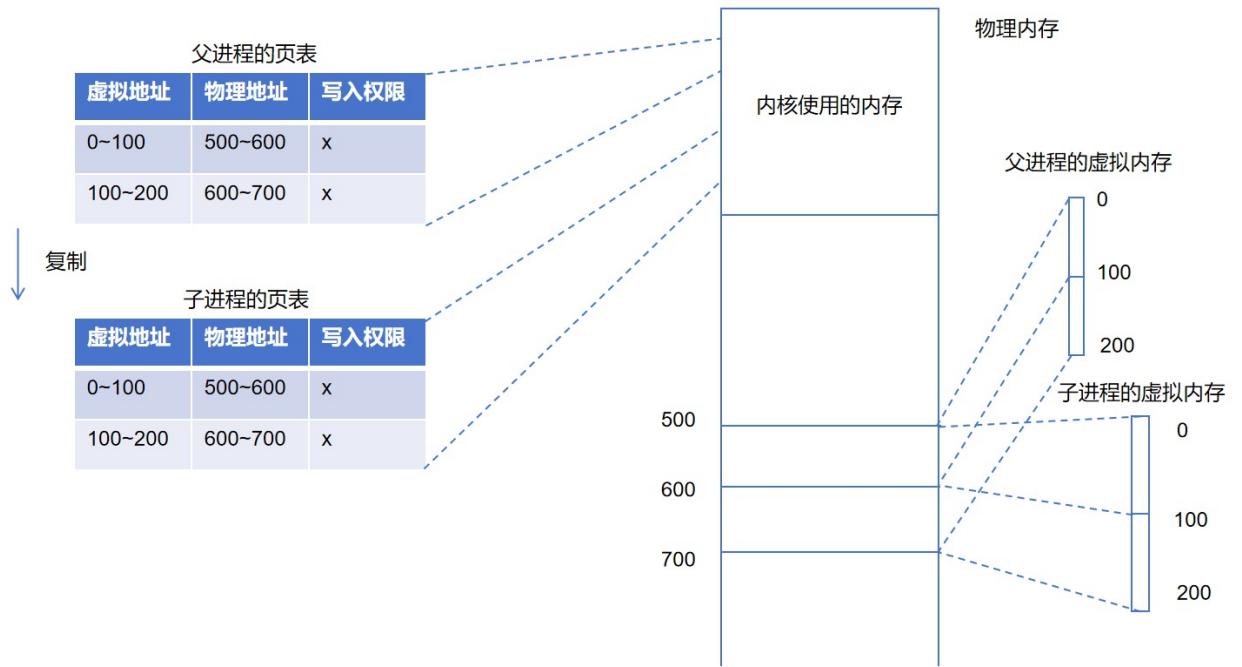
- **x86** 架构上(32位)，每个进程虚拟地址空间只有 **4GB**，数据库之类的大型程序在此架构容易引发虚拟内存不足。
- **x86_64** 架构上(64位)，虚拟地址空间被扩充到 **128TB**

由于 **128TB** 的虚拟地址空间，虚拟内存不足显得非常罕见。

物理内存不足与进程的虚拟内存剩余多少无关。

5.3 写时复制

之前提到 `fork()` 将父进程的所有内存直接复制给子进程，现在使用虚拟内存机制，我们可以仅仅将父进程页表复制给子进程。虽然此时父子进程的页表项都有标有写入权限的字段，但是此时二者都无法进行写入：



此后，如果父子进程都只进行读取操作，那么父子进程就可以共享这块物理内存(节省内存)；当有一方打算修改这部分的数据，按照下述流程解除共享：

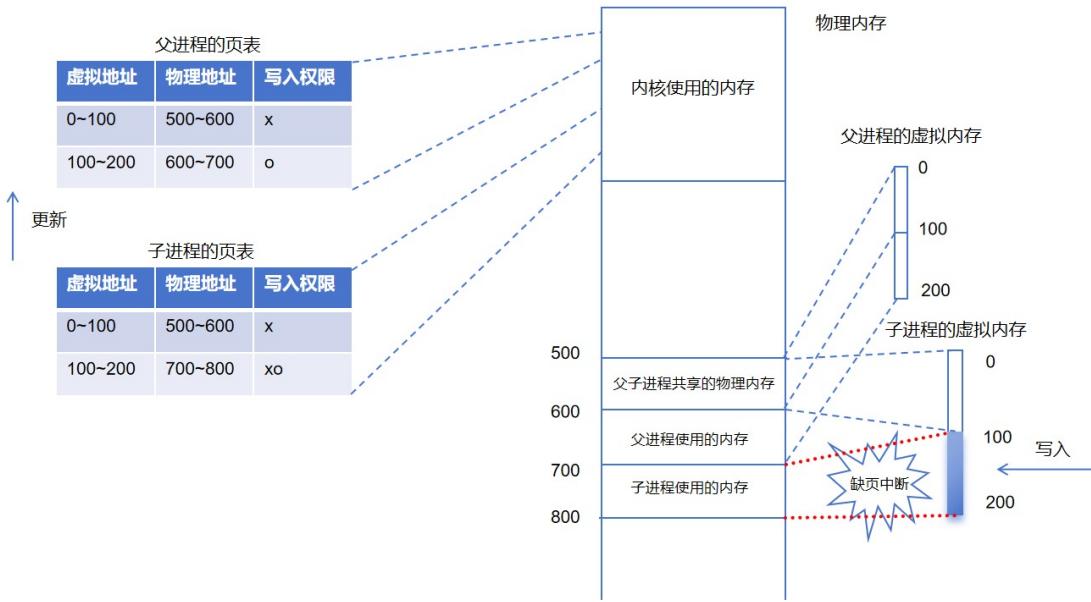
- 因为没有写入权限，尝试写入时，cpu 触发缺页中断
- cpu 转换到内核模式，缺页中断机构开始运行
- 对于被访问的页面，缺页中断机构将复制一份放到别的地方，然后为其分配给尝试写入的进程，并根据请求更新其中的内容。
- 为父进程和子进程双方更新与已解除共享的页面对应的页表项
 - 对于执行写入操作的一方，将其页表项重新连接到新分配的物理页面，并赋予写入权限
 - 对于另一方，也只需对其页表项重新赋予写入权限即可

物理内存并非在发起 `fork()` 系统调用时进行复制，而是在尝试写入时复制，所以这个机制称为 **写时复制**

因此，即使调用 `fork()` 成功，如果引发缺页中断的时候，没有充足的物理内存，也会引发内存不足的问题。

实验

下面，我们通过实验来观察发生写时复制的情形，需要确认：



- 在从调用 `fork()` 到开始写入的这段时间，内存区域是否由父子进程共享
- 在向内存区域执行写入时，是否引发缺页中断。

因此，我们需要编写实现下述要求的程序：

- 获取 **100MB** 内存，并访问所有页面
- 确认系统的内存使用量
- 调用 `fork()` 系统调用
 - 父进程和子进程分别执行以下处理
 - 父进程：等待子进程结束运行
 - 子进程：
 - 显示系统的内存使用量以及自身虚拟内存使用量，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
 - 访问申请到的 **100MB** 的所有空间
 - 再次显示系统的内存使用量以及自身的虚拟内存使用量，物理内存使用率，硬性页缺失发生次数和软性页缺失发生次数

grep 正则表达式：

```
ps -eo pid,comm | grep '^ *[pid] ' : 可以匹配以 pid 开头的一行的文本
```

cow.cpp

```
#include <sys/wait.h>
#include <unistd.h>
#include <iostream>
#include <cstring>
#include <err.h>

constexpr int BUFFER_SIZE = 100 * 1024 * 1024;
constexpr int PAGE_SIZE = 4096;
constexpr int COMMAND_SIZE = 4096;

char *p;
char command[COMMAND_SIZE];

void child_fn() {
    std::cout << "*** child ps info before memory access ***:" <<
    std::endl;
    sprintf(command, COMMAND_SIZE,\n
            "ps -eo pid,comm,vsz,rss,min_flt,maj_flt | grep '^"
            "%d ''", getpid());

    system(command);
    std::cout << "*** free memory info before memory access ***:" <<
    std::endl;
    system("free");

    // 按页为单位进行写时复制
    for (int i = 0; i < BUFFER_SIZE; i += PAGE_SIZE) {
        p[i] = 1;
    }
    std::cout << "*** child ps info after memory access ***:" <<
    std::endl;
    system(command);

    std::cout << "*** free memory info after memory access ***:" <<
    std::endl;
    system("free");
    exit(EXIT_SUCCESS);
}
```

```

}

void parent_fn() {
    wait(NULL), exit(EXIT_SUCCESS);
}

int main() {
    // 获取 `100MB` 内存，并访问所有页面
    p = (char*)malloc(BUFFER_SIZE);
    if (!p) {
        err(EXIT_FAILURE, "malloc() failed");
    }
    for (int i = 0; i < BUFFER_SIZE; i += PAGE_SIZE) {
        p[i] = 0;
    }
    std::cout << "*** free memory info before fork ***:" << std::endl;
    system("free");

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

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

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

```

编译运行后结果为：

```

syz@syz:~/projects/class5$ ./cow
*** free memory info before fork ***:
              total        used         free      shared  buff/cache available
Mem:       7942516       650164     6880448          3352       411904
7056212
Swap:      2097152           0     2097152
*** child ps info before memory access ***:

```

```

# pid & comm & vsz & rss & min_flt & maj_flt
1366 cow          108464 103684   29      0
*** free memory info before memory access ***:
              total        used        free      shared  buff/cache
available
Mem:       7942516       650664     6879948       3352      411904
7055712
Swap:      2097152           0     2097152
*** child ps info after memory access ***:
# pid & comm & vsz & rss & min_flt & maj_flt
1366 cow          108464 103684 25629      0
*** free memory info after memory access ***:
              total        used        free      shared  buff/cache
available
Mem:       7942516       752724     6777888       3352      411904
6953652
Swap:      2097152           0     2097152

```

可以看到：

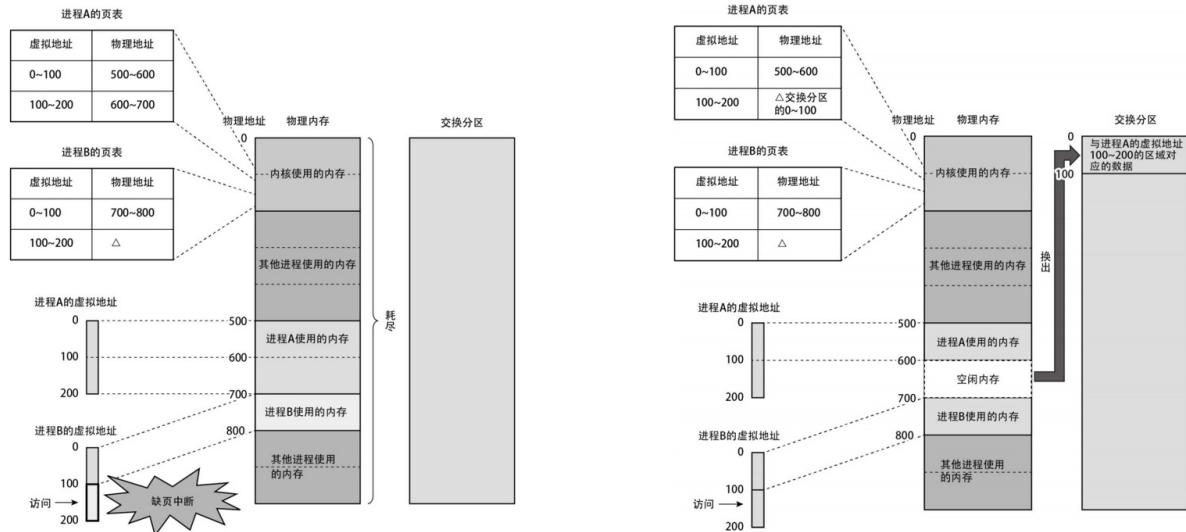
- 子进程在没有修改共享物理内存空间的内容时，内存使用量一直都是 **650164KB**（系统还在运行其它程序），当子进程访问并修改所有页面后(总共 **100MB** 的内容)，系统内存使用量约为 **99.66796875MB**
- 当子进程访问并且修改共享物理内存的内容时，发生 **min_flt** 的次数由 **29** 上升到 **25629**
- 子进程通过 **ps -eo** 输出的 **vsz,rss** 没有变化，是因为子进程并没有进行申请内存操作。

5.4 Swap

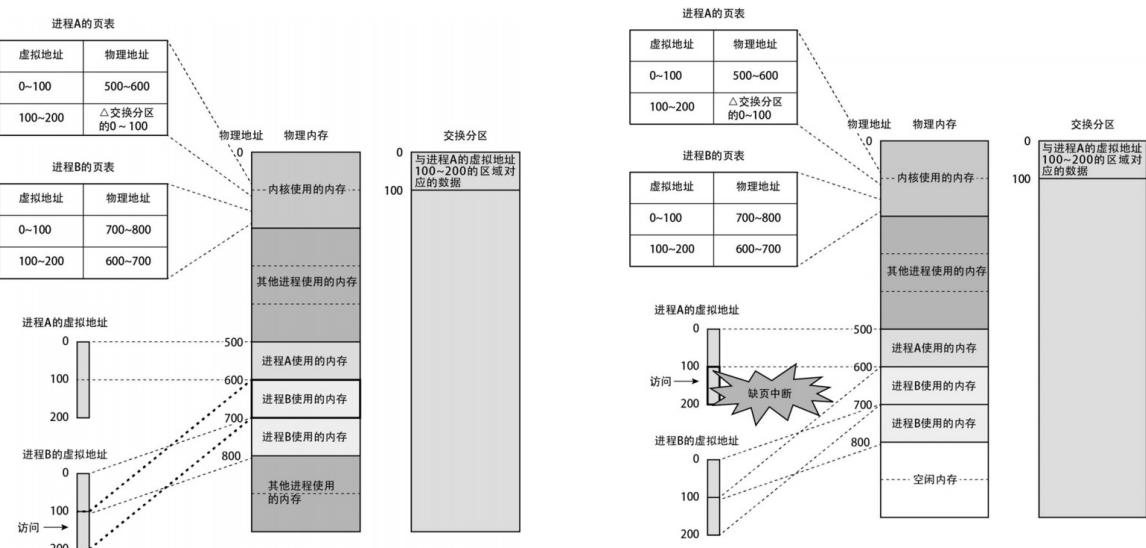
当物理内存耗尽，系统会进入 **OOM** 状态，Linux 提供了一个针对 **OOM** 的补救措施，即 **Swap**。

这一功能使得我们可以将外部存储器的一部分容量暂时当作内存使用。当物理内存不足的情况下，出现获取物理内存的申请时，物理内存的一部分页面保存到外部存储器中，这里用于保存页面的区域称为 **交换分区** (Windows 系统这被称为 **虚拟内存**)，交换分区由管理员在构建系统时设置。

当物理内存不足时候，此时如果需要更多的物理内存，触发缺页中断，然后通过某种算法确定换出内存的页区，将其移外存中。这里显示被换出页面在交换分区的上的地址信息记录在页表项里，实际上是记录在内核中专门用于管理交换分区的区域上。



然后多出来的空闲内存分配给进程 **B**，接着当程序释放了一些进程的内存后，如果进程 **A** 对于先前保存到交换分区的页面发起访问，就会发生缺页中断：



接着执行 **换入**：

程序必须放在内存上然后加载到 CPU 上执行，因此当你需要运行某一段，而物理内存不足，此时会将一个或者多个现在暂时不需要的页面换出到 Swap，等你需要运行被

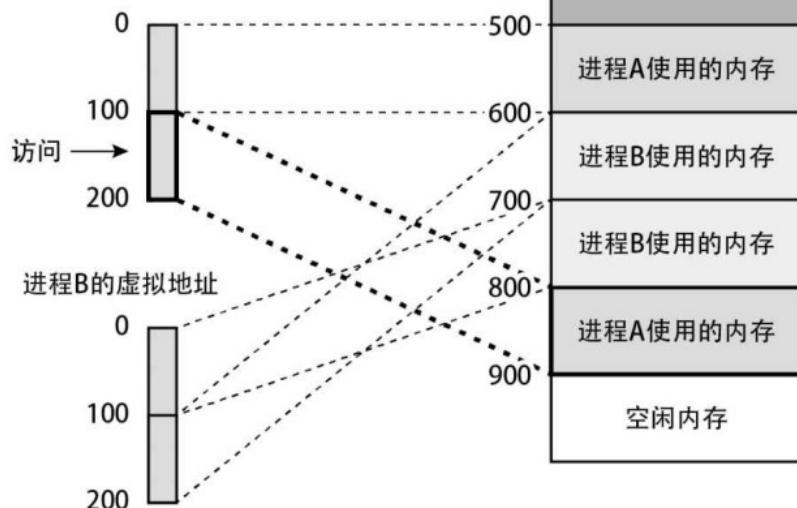
进程A的页表

虚拟地址	物理地址
0~100	500~600
100~200	800~900

进程B的页表

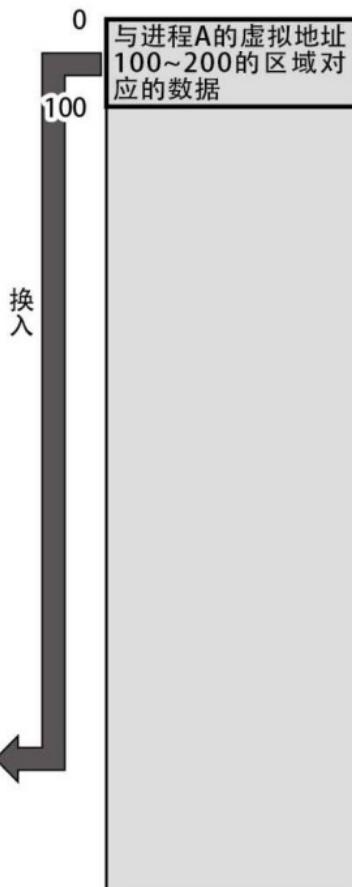
虚拟地址	物理地址
0~100	700~800
100~200	600~700

进程A的虚拟地址



物理地址 物理内存

交换分区



换出的那一段程序，(如果物理内存仍然不足) 它会将内存中现在暂时不需要的页面换出到 Swap，将你现在需要运行的程序换入到内存中。

Swap 看起来是一个将可用内存量扩充为 **实际搭载内存 + 交换分区内存**，但是相比于内存的访问速度，对普通外部存储器的访问速度慢了几个数量级。

如果物理内存不足，就容易发生 **系统抖动**，系统暂时无法响应。

实验

通过 **swapon --show** 可以查看系统交换分区的信息

```
syz@syz:~/projects/class5$ swapon --show
NAME      TYPE      SIZE USED PRIO
/dev/sdb  partition 2G    0B   -2
```

交换分区内存大小约为 **2GB**

还可以通过 `free` 查看交换分区的内存：

```
syz@syz:~/projects/class5$ free
total        used         free      shared  buff/cache   available
Mem:    7942516     500416    7039820          3300    402280    7206768
Swap:  2097152          0    2097152
```

系统运行期间，定期通过 `sar -W` 可以查看系统中是否发生了交换处理。

```
syz@syz:~/projects/class5$ sar -W 1 1
Linux 5.15.133.1-microsoft-standard-WSL2 (syz)  03/03/24           _x86_64_      (20 CPU)

14:32:34      pswpin/s pswpout/s
14:32:35          0.00      0.00
Average:        0.00      0.00
```

`pswpin`:每sec发生换入的次数, `pswpout`:每sec发生换出的次数

使用 `sar -S 1` 可以确认该交换处理是暂时的，还是毁灭性的：

```
syz@syz:~/projects/class5$ sar -S 1 1
Linux 5.15.133.1-microsoft-standard-WSL2 (syz)  03/03/24           _x86_64_      (20 CPU)

14:35:38      kbswpfree kbswpused  %swpused   kbswpcad  %swpca
14:35:39      2097152          0      0.00          0      0.00
Average:      2097152          0      0.00          0      0.00
```

当 `kbswpused` 字段值了解交换分区使用量的变化趋势，如果这个值增加就会很危险。

这里补充一下缺页中断的知识：

- **硬性页缺失**：需要访问外部存储器的缺页中断
- **软性页缺失**：无需访问外部存储器的缺页中断

二者都需要经过内核进行处理，硬性页缺失产生的影响越大。

5.5 多级页表

`x86_64` 架构上，虚拟地址空间一般大小为 `128TB`，页面大小为 `4KB`，页表项大小为 `8B`，那么创建一个页表就需要：

$$8B \times 128TB / 4KB = 256GB$$

正常来说现在一般计算机内存只有 `16G/32G` 左右，因此这样一个进程也创建不了。

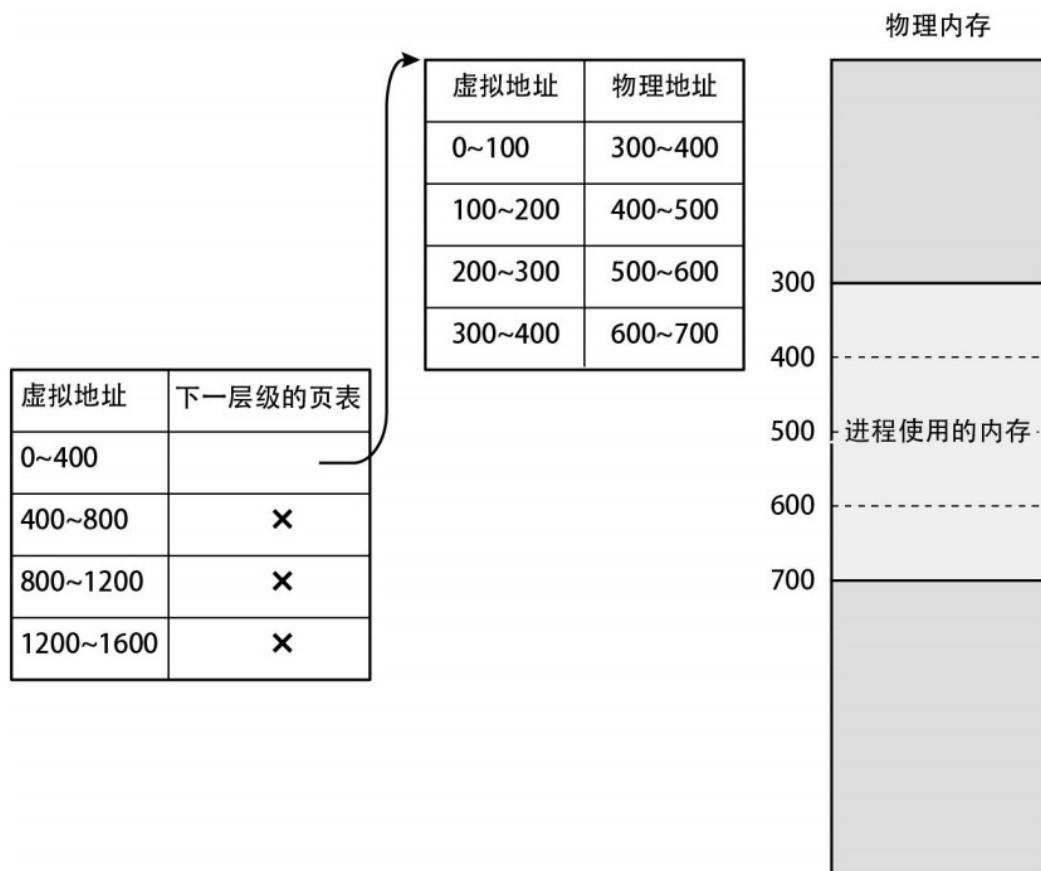
`x86_64` 架构采用多级页表避免了这种情况。

如下，我们使用一个简单的模型来介绍多级页表：假设一个页面大小为 `100B`，虚拟地址空间为 `1600B`，这种情况如果采用单层页表，结果为：

假设多级页表为每 4 个页面归为 1 组的 2 级结构，如图：

物理内存

虚拟地址	物理地址
0~100	300~400
100~200	400~500
200~300	500~600
300~400	600~700
400~500	x
500~600	x
600~700	x
700~800	x
800~900	x
900~1000	x
1000~1100	x
1100~1200	x
1200~1300	x
1300~1400	x
1400~1500	x
1500~1600	x



随着虚拟内存使用量增加到一定程度，多级页表的内存使用量就会超过单层页表，但这种情况比较罕见。[x86_64](#) 的页表结构达到了 4 级，这里不涉及。

可以通过 `sar -r ALL` 命令中的 `kbpgtbl` 查看 **页表使用的内存量**

`kbpgtbl (kb-pg-tbl)` : 表示当所有进程页表使用的内存量(KB)

```
syz@syz:~/projects/class5$ sar -r ALL 1 1
Linux 5.15.133.1-microsoft-standard-WSL2 (syz) 03/03/24          _x86_64_          (20 CPU)

15:01:14    kbmemfree   kbavail   kbmused   %memused   kbbuffers   kbcached   kbcommit   %commit   kbactive   kbinact
  kbdirty   kbanonpg   kbslab   kbkstack   kbpgtbl   kbvmused
15:01:15      6972472    7155700    484836     6.10      15888     368936     836968      8.34     165308     442832
  0       218100     100384     5168      3428      26504
Average:    6972472    7155700    484836     6.10      15888     368936     836968      8.34     165308     442832
  0       218100     100384     5168      3428      26504
```

5.6 标准大页

随着进程虚拟内存不断增加，进程页表使用的物理内存量也会增加。

此时，除了内存使用量增加的问题外，还存在 `fork()` 系统调用的执行速度变慢的问题，因为 `fork()` 写时复制需要为子进程复制一份和父进程同样大小的页表。为了解决这个问题，Linux 提供了标准大页。

例如：

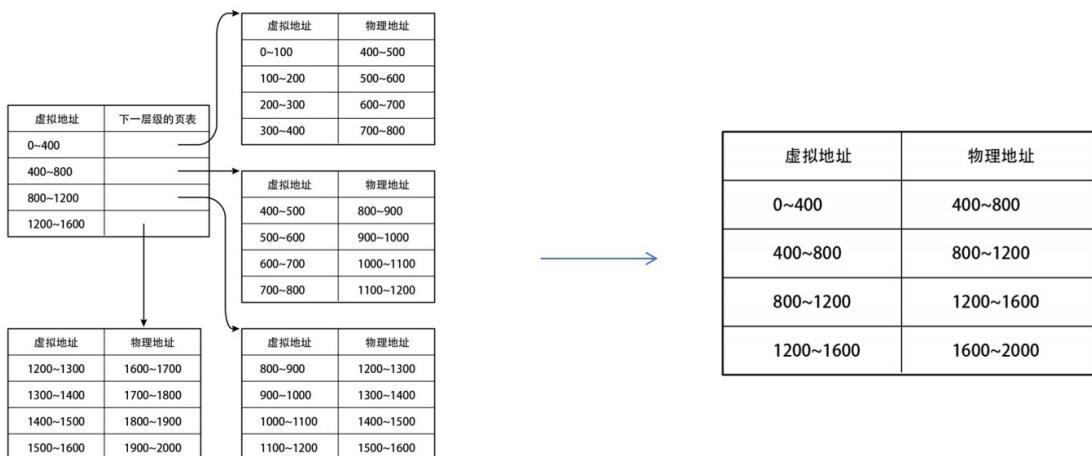


图 5-47 所有页面都被分配物理内存时的情形

可以看到，页表项的数量由 20 个减小到 4 个，将普通页面置换成标准大页，不但能降低页表的内存使用量，还能提高 `fork()` 系统调用的执行速度。

如何使用

C/C++ 中可以：

```
char *addr = (char*)mmap(NULL, 2 * 1024 * 1024, PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_HUGETLB, fd, 0);
```

透明大页

当虚拟地址空间内连续多个 4KB 的页面符合特定条件时，通过透明大页机制能将它们自动换成一个大页。

这虽然看起来非常便利，但是存在一些问题：将多个页面汇聚成一个大页的处理，以及不满足上述条件时，将大页重新拆分成多个 4KB 的页面的处理，会引起局部性能下降。因此搭建系统时候，有时会选择禁用透明大页。

```
syz@syz:~/projects/class5$ cat /sys/kernel/mm/transparent_hugepage/enabled  
[always] madvise never
```

希望禁用该功能，只需要往该文件写入 `never`

```
syz@syz:~/projects/class5$ sudo su // 切换到超级用户  
[sudo] password for syz:  
root@syz:/home/syz/projects/class5# echo never  
>/sys/kernel/mm/transparent_hugepage/enabled  
root@syz:/home/syz/projects/class5# exit // 退出  
exit  
syz@syz:~/projects/class5$ cat /sys/kernel/mm/transparent_hugepage/enabled  
always madvise [never]
```

设定为 `madvise` 时候表示仅对由 `madvise()` 系统调用设定的内存区域启用透明大页机制。