一文读懂共享内存

本文主题为共享内存，但是所关联的基础知识也会过一下，希望对大家有所帮助。

（共享内存并不是什么新技术，而是计算机操作系统的基础知识，这里带着大家复习回顾下）

一、前言知识

1. 定义

· 硬件定义

> **共享内存**指在多处理器的计算机系统中，可以被不同中央处理器CPU访问的大容量内存。由于多个CPU需要快速访问存储器，这样就要对存储器进行缓存。由于其他处理器可能也要存取，任一缓存数据更新后，共享内存就需要立即更新，否则不同处理器可能用到不同的数据。

· 软件定义

> **共享内存**指可被多个线程或者进程存取的内存

这种定义是比较宽泛的，主要分为多线程和多进程

多线程环境

多线程环境（多处理器）中数据共享会有线程安全的问题，线程修改共享数据需要加锁，来保证操作的原子性；

比如百度百科上关于Int64的说明

> 警告：在 32 位 Intel 计算机上分配 64 位值不是原子操作；即该操作不是线程安全的。这意味着，如果两个人同时将一个值分配给一个静态 Int64 字段，则该字段的最终值是无法预测的。

32位计算机上，对Int64类型的变量赋值或取值，在CPU指令的层级，是两步操作，分别写或读内存的前32位和后32位。如果在这2个CPU指令之间，其他线程对这个变量进行读写，就会发生并发问题。例如：线程1先写了变量的前32位，这时线程2写了变量的前32位和后32位，接下来线程1写了变量的后32位。最终结果，内存的前32位是线程2写的，后32位是线程1写的。最终内存中的数据是一个完全错误的数据。

多进程环境

在同一台物理机器多进程环境中，共享内存是进程间通信(Inter-Process Communication)的一种方式，原理就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。其申请分配的内存由操作系统管理，内存数据不会随着进程的关闭而回收，除非强制调用相关函数代码(remove)或者系统命令来回收(linux ipcrm -m)。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。

linux平台可以使用ipcs命令查看系统共享内存使用情况，在系统目录/dev/shm/下一般会看到有共享内存映射文件

```
3048 ○  ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 0          zabbix     600        365056     6          dest

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x00000000 65536      zabbix     600        14
```

由上图可以看出，可以到当前系统zabbix进程在使用共享内存

说明：共享内存一般特指多进程共享内存（Shared Memory）

2. 进程通信方式

· 匿名管道通信

匿名管道(pipe)：管道是一种半双工的通信方式，数据只能**单向流动**，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指**父子进程关系**。

· 高级管道通信

高级管道(popen)：将另一个程序当做一个新的进程在当前程序进程中启动，则它算是当前程序的子进程，这种方式我们成为高级管道方式。
这个在python中比较多，有对应的API。

· 命名管道通信

命名管道 (named pipe)： 命名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

· 流管道

流管道s_pipe: 去除了第一种限制,可以双向传输。

· 消息队列通信

消息队列(message queue)： 消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

· 信号量通信

信号量(semophore)： 信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

· 信号

信号 (sinal)： 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。主要作为进程间以及同一进程不同线程之间的同步手段。

· 共享内存通信

共享内存(shared memory)：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。

· 套接字通信

套接字(socket)： 套接口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。

3. 通信方式比较

各种通信方式的比较和优缺点

- 管道：速度慢，容量有限，只有父子进程能通讯
- FIFO：任何进程间都能通讯，但速度慢
- 消息队列：容量受到系统限制，且要注意第一次读的时候，要考虑上一次没有读完数据的问题
- 信号量：不能传递复杂消息，只能用来同步
- 共享内存区：能够很容易控制容量，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全，当然，共享内存区同样可以用作线程间通讯，不过没这个必要，线程间本来就已经共享了同一进程内的一块内存

如果用户传递的信息较少或是需要通过信号来触发某些行为，软中断信号机制不失为一种简捷有效的进程间通信方式；但若是进程间要求传递的信息量比较大或者进程间存在交换数据的要求，那就需要考虑别的通信方式了。

无名管道简单方便．但局限于单向通信的工作方式．并且只能在创建它的进程及其子孙进程之间实现管道的共享；有名管道虽然可以提供给任意关系的进程使用。但是由于其长期存在于系统之中，使用不当容易出错．所以普通用户一般不建议使用。

消息缓冲可以不再局限于父子进程，而允许任意进程通过共享消息队列来实现进程间通信，并由系统调用函数来实现消息发送和接收之间的同步，从而使得用户在使用消息缓冲进行通信时不再需要考虑同步问题，使用方便但是信息的复制需要额外消耗CPU的时间，不适宜于信息量大或操作频繁的场合。

共享内存针对消息缓冲的缺点改而利用内存缓冲区直接交换信息，无须复制，快捷、信息量大是其优点。但是共享内存的通信方式是通过将共享的内存缓冲区直接附加到进程的虚拟地址空间中来实现的，因此，这些进程之间的读写操作的同步问题操作系统无法实现。必须由各进程利用其他同步工具解决。另外，由于内存实体存在于计算机系统中，所以只能由处于同一个计算机系统中的诸进程共享。不方便网络通信。跨机器网络通讯使用套接字比较合适。

二、共享内存能解决什么问题?

这个标题简直是答非所问，共享内存当然是进程间通信啊，难道不是为了进程间通讯吗?
答案：是也不是!

利用共享内存的特性，我们可以做很多事情

- 多进程共享内存，我们可以用来进程间传递数据，比如主从机制等；
- 共享内存由操作系统管理，不会随着进程消失而消失，除非你想让消失，我们可以利用这个特性对有状态进程进行数据容灾，内存中数据不会随着突然宕机而丢失，当进程再次启动的时候还可以急需使用共享内存数据

1. 我们为什么要引入它?

在我们服务器架构中，或多或少存在一些有状态的Server（比如DBServer PublicServer等），即内存中存在一些角色重要数据。并且由于数据库IO以及网络压力等问题，内存数据更新后不能实时落地，一般策略为脏数据定时入库。

由于这种定时入库机制存在，当内存数据更新后正式入库前进程异常crash后，这期间修改的数据就会丢失，造成严重后果，主要在public和dbserver上比较常见。我们可以使用共享内存的方式，将每次更新后数据放入共享内存中，进程宕机数据并不会丢失。

以上是引入共享内存的初衷：进程宕机容灾

2. 具体应用场景

PublicServer内存数据容灾

Publicserver进程区服内单点，主要用于存放当前区服内角色全局数据，属于关键节点如果宕机或者停止服务，那么将会造成严重后果

- 更新数据的时候先更新普通内存数据，然后将更新后的数据保存到共享内存
- 进程宕机后重启后,基本数据加载玩后，再用共享内存数据把普通内存数据覆盖
- 如果进程是正常关闭的（kill -15信号），那么将共享内存回收给操作系统

Publicserver主从机制完善

share memory 设计初衷就是进程间通信，因此我们可以使用一些同步机制wait和notify，来保证master和slave进程数据同步。前提是我们publicserver部署在同一个机器上。
具体实现方案待讨论。

3. DBServer内存缓存数据容灾

DBServer服务器上存放着角色全量数据，必须考虑容灾方案。但是属于跨机器多进程部署的，使用共享内存容灾始终有缺陷，不是很完美，经过讨论最终方案被否决。

可以看出，Shm不能跨机器的特点导致在现在很多分布式架构系统应用比较少见

三、具体实现与应用

1. Unix && Unix-like

当我们在linux系统中进行进程间通信时，会发现例如共享内存，信号量，消息队列等方式时，会发现有System v以及POXIS两种类型。带着大家看下资料介绍：

> **POSIX(Portable Operating System Interface for Computing Systems)**是由IEEE 和ISO/IEC 开发的一簇标准。该标准是基于现有的UNIX 实践和经验，描述了操作系统的调用服务接口，用于保证编制的应用程序可以在源代码一级上在多种操作系统上移植运行。（此处省略UINT_MAX个字符）

> POSIX(Portable Operating System Interface)可移植操作系统接口，这样的简写完全是为了和UNIX读起来更像而已。 目前POSIX已经成为类UNIX（Unix-like）操作系统编程的通用接口，极大方便了类UNIX环境下应用程序源码级的可移植性。Glibc(GNU C Library),即C运行库，是Linux系统中最底层的API，它就是完全按照POSIX标准编写的。
>
> **System V**， 曾经也被称为 AT&T System V，是Unix操作系统众多版本中的一支，就是当年UNIX厂家混战中，比较强大的一个诸侯王。它最初由 AT&T 开发，在1983年第一次发布。一共发行了4个 System V 的主要版本：版本1、2、3 和 4。System V Release 4，或者称为SVR4，是最成功的版本，成为一些UNIX共同特性的源头，例如 ” SysV 初始化脚本 “ (/etc/init.d)，用来控制系统启动和关闭，System V Interface Definition (SVID) 是一个System V 如何工作的标准定义。
>
> AT&T 出售运行System V的专有硬件，但许多(或许是大多数)客户在其上运行一个转售的版本，这个版本基于 AT&T 的实现说明。流行的SysV 衍生版本包括 Dell SVR4 和 Bull SVR4。当今广泛使用的 System V 版本是 SCO OpenServer，基于 System V Release 3，以及SUN Solaris 和 SCO UnixWare，都基于 System V Release 4。
>
> SUN公司大家应该都知道，依然是现在商用服务器操作系统重要提供商，但是我们常用的Linux操作系统并不是基于此的，但是这里要感谢POSIX这样标准化的努力，是它兼容了绝大部分System V的规格，减少了各类操作系统之间移植的麻烦。

总结：POSIX为IEEE定义的标准，可移植性好，而System V是早期最强的一个版本。
一般的，Linux/Unix系统编程中都会支持System V和POXIS，当然实现原理以及两者函数API接口也不一样。

MMAP内存映射文件

内核怎样保证各个进程寻址到同一个共享内存区域的内存页面?

- ***Page Cache 与 Swap Cache***： *page cache*是与文件映射对应的，而*swap cache*是与匿名页对应的。如果一个内存页面不是文件映射，则在换入换出的时候加入到*swap cache*，如果是文件映射，则不需要交换缓冲。一个被访问文件的物理页面都驻留在*page cache*或*swap cache*中，一个页面 的所有信息由*struct page*来描述。*struct page*中有一个域为指针***mapping*** ，它指向一个***struct address_space***类型结构。*page cache*或*swap cache*中的所有页面就是根据*address_space*结构以及一个偏移量来区分的。
- **文件与address_space结构的对应**：一个具体的文件在打开后，内核会在内存中为之建立一个**struct inode**结构，其中的i_mapping域指向一个address_space结构。这样，一个文件就对应一个address_space结构，一个address_space与一个偏移量能够确定一个page cache 或swap cache中的一个页面。因此，当要寻址某个数据时，很容易根据给定的文件及数据在文件内的偏移量而找到相应的页面。

```
struct address_space {
       struct inode           *host;            /* owner: inode, block_device拥 有它的节点 */
       struct radix_tree_root    page_tree;       /* radix tree of all pages包含全部页面的radix树 */
       rwlock_t        tree_lock;       /* and rwlock protecting it保护page_tree的自旋锁 */
       unsigned int           i_mmap_writable;/* count VM_SHARED mappings共享映射数 VM_SHARED记数*/
       struct prio_tree_root      i_mmap;         /* tree of private and shared mappings优先搜索树的树根*/
       struct list_head       i_mmap_nonlinear;/*list VM_NONLINEAR mappings 非线性映射 的链表头*/
       spinlock_t              i_mmap_lock; /* protect tree, count, list 保 护i_mmap的自旋锁*/
       unsigned int           truncate_count;      /* Cover race condition with truncate将文件截断的记数*/
       unsigned long         nrpages;  /* number of total pages 页总数*/
       pgoff_t                  writeback_index;/* writeback starts here 回写的起始偏移*/
       struct address_space_operations *a_ops;     /* methods  操作函数表*/
       unsigned long         flags;            /* error bits/gfp mask , gfp_mask掩码与错误标识 */
       struct backing_dev_info *backing_dev_info; /* device readahead, etc预读信息 */
       spinlock_t              private_lock;   /* for use by the address_space  私有address_space锁*/
       struct list_head       private_list;     /* ditto 私有address_space链表*/
       struct address_space     *assoc_mapping;    /* ditto 相关的缓冲*/
} __attribute__((aligned(sizeof(long))));
```

- **进程调用mmap()时**，只是在进程空间内新增了一块相应大小的缓冲区，并设置了相应的访问标识，但并没有建立进程空间到物理页面的映射。因此，第一次访问该空间时，会引发一个缺页异常。
  - **对于共享内存映射情况**，缺页异常处理程序首先在swap cache中寻找目标页（符合address_space以及偏移量的物理页），如果找到，则直接返回地址；如果没有找到，则判断该页是否在交换区(swap area)，如果在，则执行一个换入操作；如果上述两种情况都不满足，处理程序将分配新的物理页面，并把它插入到page cache中。进程最终将更新进程页表。

- **对于映射普通文件情况（非共享映射）**，缺页异常处理程序首先会在page cache中根据address_space以及数据偏移量寻找相应的页面。如果没有找到，则说明文件数据还没有读入内存，处理程序会从磁盘读入相应的页面，并返回相应地址，同时，进程页表也会更新。

所有进程在映射同一个共享内存区域时，情况都一样，在建立线性地址与物理地址之间的映射之后，不论进程各自的返回地址如何，实际访问的必然是同一个共享内存区域对应的物理页面。
注：一个共享内存区域可以看作是特殊文件系统shm中的一个文件，shm的安装点在交换区上。

mmap属于linux系统调用，使得进程之间通过映射同一个普通文件实现共享内存。

mmap函数主要的功能就是将文件或设备映射到调用进程的地址空间中，当使用mmap映射文件到进程后,就可以直接操作这段虚拟地址进行文件的读写等操作,**不必再调用read，write等系统调用**。在很大程度上**提高了系统的效率和代码的简洁性**。

使用mmap函数的主要目的是：
- 对普通文件提供内存映射I/O，可以提供无亲缘进程间的通信；
- 提供匿名内存映射，以供亲缘进程间进行通信。
- 对shm_open创建的POSIX共享内存区对象进程内存映射，以供无亲缘进程间进行通信。

mmap相关接口说明

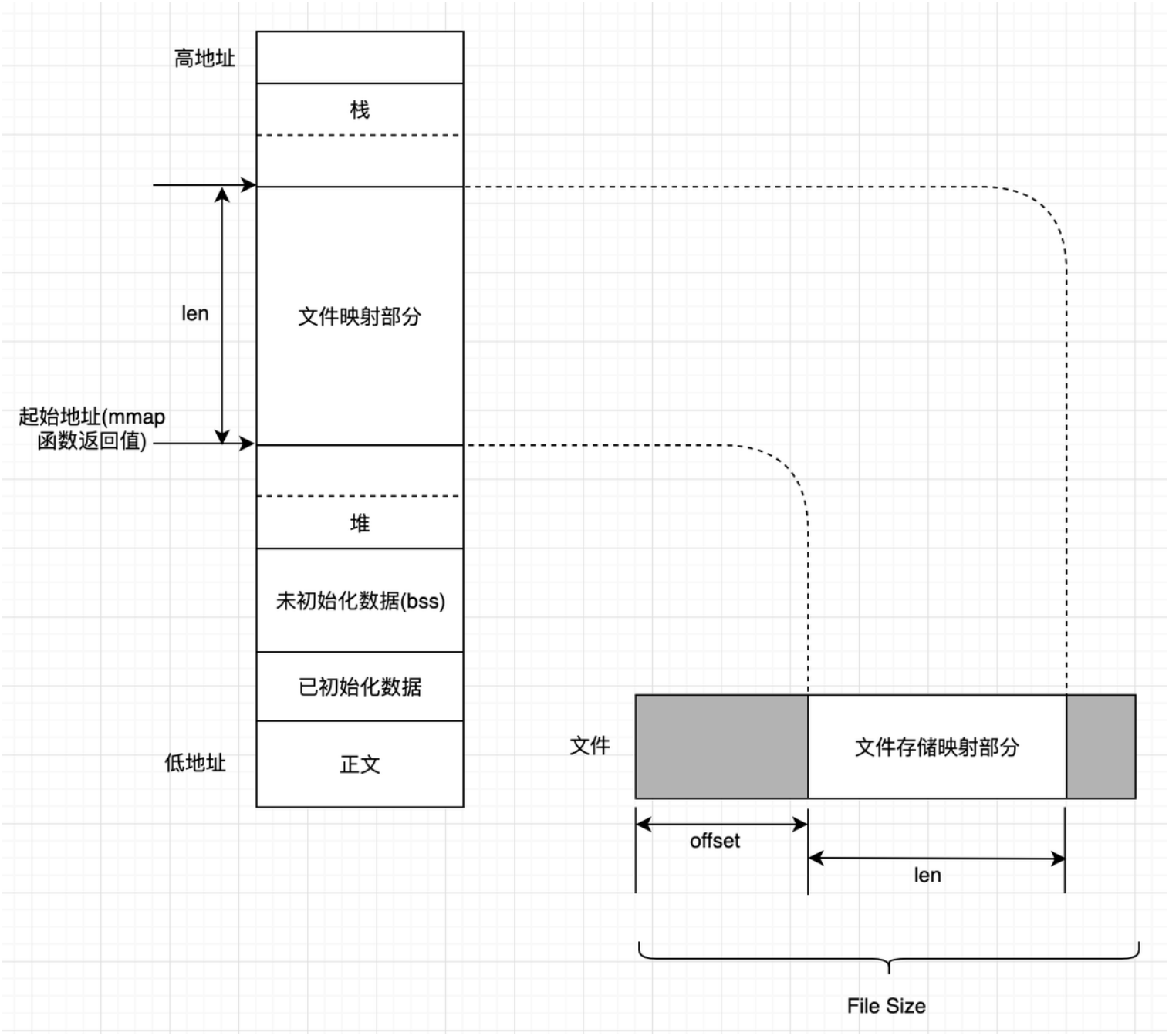
```
 #include <sys/mman.h>
//start:设置需要映射的固定内存地址，一般为NULL，系统来自动分配
//len: 映射长度
//prot: 映射区域的权限PROT_READ、PROT_WRITE、PROT_EXEC、PROT_NONE：数据不可访问
//flags: 设置内存映射区的类型标志:MAP_SHARED、MAP_PRIVATE、MAP_FIXED
//fd: 文件描述符
//offset: 文件偏移量，偏移offset后面的文件才会被映射到
void *mmap(void *start, size_t len, int prot, int flags, int fd, off_t offset)
//成功返回映射到进程地址空间的起始地址，失败返回MAP_FAILED
//mmap成功后，可以关闭fd，一般也是这么做的，这对该内存映射没有任何影响。
```

```
#include <sys/mman.h>
//从进程的地址空间中删除一个映射关系
//参数同mmap函数
int munmap(void *start, size_t len);
```

```
5 //成功返回0，出错返回-1
```

对于一个MAP_SHARED的内存映射区，内核的虚拟内存算法会保持内存映射文件和内存映射区的同步，也就是说，对于内存映射文件所对应内存映射区的修改，内核会在稍后的某个时刻更新该内存映射文件。如果我们希望硬盘上的文件内容和内存映射区中的内容实时一致，那么我们就可以调用msync开执行这种同步：

```
1 #include <sys/mman.h>
2 /* flags：同步标志，有一下三个标志：
3 //  MS_ASYNC：异步写，一旦写操作由内核排入队列，就立刻返回；
4 //  MS_SYNC：同步写，要等到写操作完成后才返回。
5 //  MS_INVALIDATE：使该文件的其他内存映射的副本全部失效。
6 int msync(void *start, size_t len, int flags);
7 //成功返回0，出错返回-1
```



内存映射文件例子

内存映射区的大小

Linux下的内存是采用页式管理机制。通过mmap进行内存映射，**内核生成的映射区的大小都是以页面大小PAGESIZE为单位，即为PAGESIZE的整数倍**。如果mmap映射的长度不是页面大小的整数倍，那么多余空间也会被闲置浪费。默认页大小为4k（4096B）

查看当前linux系统页大小：

```
1 #include <iostream>
```

```cpp
#include <unistd.h>
int main()
{
    std::cout<<"page size:"<<sysconf(_SC_PAGE_SIZE)<<std::endl;
    return 0;
}
```

pagesize.cpp

```cpp
#include <iostream>
#include <unistd.h>
int main()
{
    std::cout<<"page size:"<<sysconf(_SC_PAGE_SIZE)<<std::endl;
    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
huzhigen@genge: ~/work_space/codes/shm_demo $ g++ pagesize.cpp -o pagesize
huzhigen@genge: ~/work_space/codes/shm_demo $ ./pagesize
page size:4096
huzhigen@genge: ~/work_space/codes/shm_demo $
```

当内存映射大小等于文件长度情况

```cpp
#include <iostream>
#include <cstring>
#include <cerrno>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>

using namespace std;

#define  PATH_NAME "mmap_equal_file"

int main(int argc, char **argv)
```

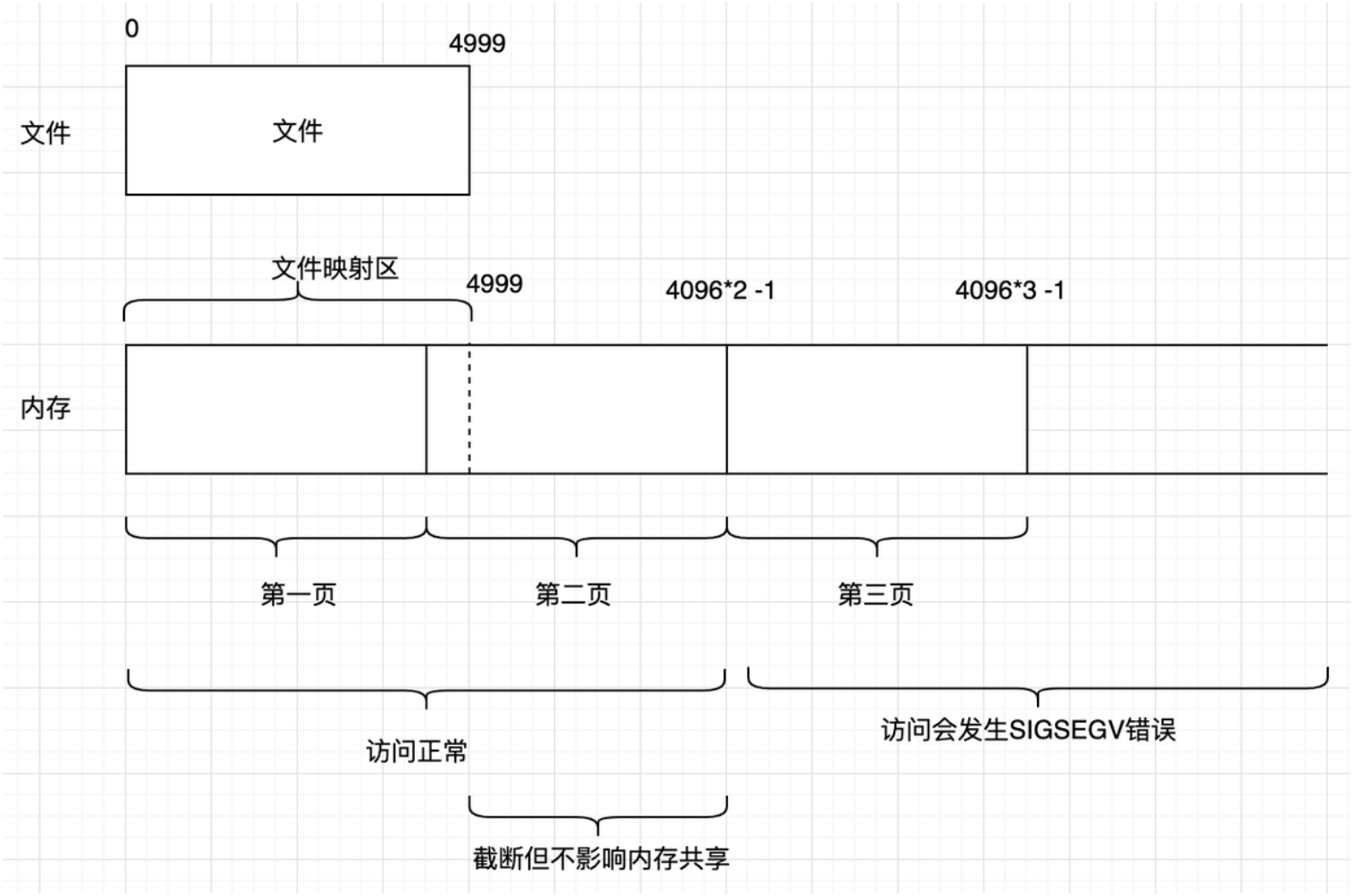
```
13 {
14     int fd;
15 
16     fd = open(PATH_NAME, O_RDWR | O_CREAT, 0666);
17     if (fd < 0)
18     {
19         cout<<"open file "<<PATH_NAME<<" failed...";
20         cout<<strerror(errno)<<endl;
21         return -1;
22     }
23 
24     if (ftruncate(fd, 5000) < 0)
25     {
26         cout<<"change file size  failed...";
27         cout<<strerror(errno)<<endl;
28 
29         close(fd);
30         return -1;
31     }
32 
33     char *memPtr;
34 
35     memPtr = (char *)mmap(NULL, 5000, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
36     close(fd);
37 
38     if (memPtr == MAP_FAILED)
39     {
40         cout<<"mmap failed..."<<strerror(errno)<<endl;
41         return -1;
42     }
43 
44     cout<<"[0]:"<<(int)memPtr[0]<<endl;
45     cout<<"[4999]:"<<(int)memPtr[4999]<<endl;
46     cout<<"[5000]:"<<(int)memPtr[5000]<<endl;
47     cout<<"[8191]:"<<(int)memPtr[8191]<<endl;
```

```
48      cout<<"[8192]:"<<(int)memPtr[8192]<<endl;
49      cout<<"[4096 * 3 - 1]:"<<(int)memPtr[4096 * 3 - 1]<<endl;
50      cout<<"[4096 * 3]:"<<(int)memPtr[4096 * 3]<<endl;
51
52      return 0;
53  }
```

Demo运行结果：

```
huzhigen@genge: ~/work_space/codes/shm_demo $ g++ mmap_equal.cpp -o mmap_equal
huzhigen@genge: ~/work_space/codes/shm_demo $ ./mmap_equal
[0]:0
[4999]:0
[5000]:0
[8191]:0
[1]    4334 segmentation fault  ./mmap_equal
FAIL: 139
huzhigen@genge: ~/work_space/codes/shm_demo $
```

结果分析：



当内存映射大小大于文件长度情况

修改第一种情况代码

```
1 memPtr = (char *)mmap(NULL, 4096 * 3, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

发生SIGBUG异常

```
huzhigen@genge: ~/work_space/codes/shm_demo $ g++ mmap_big.cpp -o mmap_big
huzhigen@genge: ~/work_space/codes/shm_demo $ ./mmap_big
[0]:0
[4999]:0
[5000]:0
[8191]:0
[1]    4643 bus error  ./mmap_big
FAIL: 138
huzhigen@genge: ~/work_space/codes/shm_demo $
```

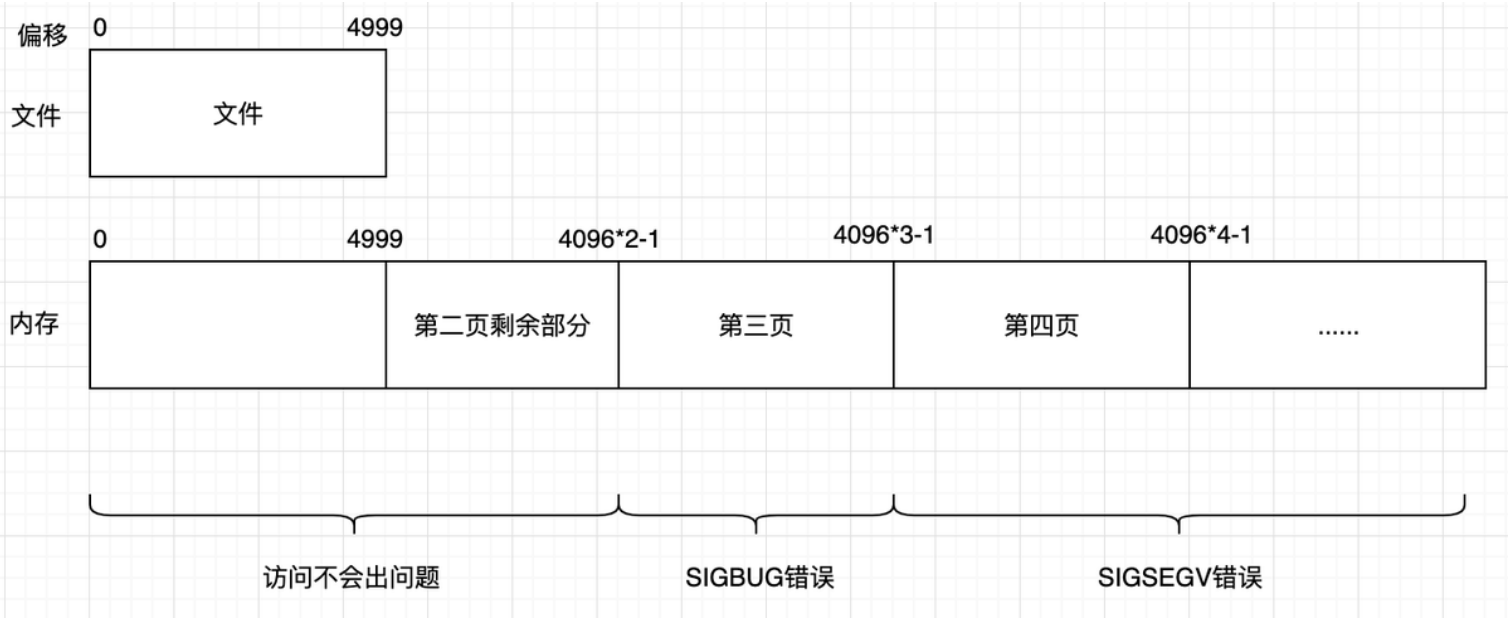
修改访问代码

```
1 cout<<"[4096 * 4]:"<<(int)memPtr[4096 * 4]<<endl;
```

```
huzhigen@genge: ~/work_space/codes/shm_demo $ g++ mmap_big.cpp -o mmap_big
huzhigen@genge: ~/work_space/codes/shm_demo $ ./mmap_big
[0]:0
[1]    4695 segmentation fault  ./mmap_big
FAIL: 139
huzhigen@genge: ~/work_space/codes/shm_demo $
```

分析与总结：

当内存映射空间大于文件映射长度时，1. 在访问内存映射区内部但超出底层支撑对象（文件）的大小的区域部分会产生SIGBUS错误 2. 访问地址大于内存映射长度时，会发生SIGSEGV错误



tips：信号SIGSEGV通常用于指示进程试图访问它不可用的存储区，比如数组越界之类的；如果映射区的每个部分不存在，那么产生SIGBUG信号。

POSIX IPC-SHM

实现剖析

多个进程调用mmap将进程空间映射到同一个文件，实际上就是大家在共享文件page cache中的内存，加快读写效率。不过文件还是会牵涉到磁盘的读写（普通文件系统），会创建普通文件，用来做共享内存显然十分笨重，所以就有了不跟磁盘扯上关系的内存文件。

POSIX 标准正是基于mmap实现的共享内存。不过是使用`shm_open`接口申请一块真实内存文件，一般会在/dev/shm/下有对应的文件被创建，使用`shm_unlink`来解除映射，其它接口与mmap保持一致。

```
#include <sys/mman.h>
//用于创建一个新的共享内存区对象或打开一个已经存在的共享内存区对象
//name：POSIX IPC的名字，前面关于POSIX进程间通信都已讲过关于POSIX IPC的规则，这里不再赘述。

//oflag：操作标志，包含：O_RDONLY，O_RDWR，O_CREAT，O_EXCL，O_TRUNC。其中O_RDONLY和O_RDWR标志必须且仅能存在一项。

//mode：用于设置创建的共享内存区对象的权限属性。和open以及其他POSIX IPC的xxx_open函数不同的是，该参数必须一直存在，如果oflag参数中没有O_CREAT标志，该位可以置0;
int shm_open(const char *name, int oflag, mode_t mode);
```

```
                                    //成功返回非负的描述符，失败返回-1

//用于删除一个共享内存区对象，跟其他文件的unlink以及其他POSIX IPC的删除操作一样，对象的析构会到对该对象的所有引用全部关闭才会发生。
int shm_unlink(const char *name);
                                    //成功返回0，失败返回-1
```

实例

写进程代码：posix_write.cpp

```
//posix_write.cpp
#include <iostream>
#include <cstring>
#include <errno.h>

#include <unistd.h>
#include <fcntl.h>
#include <semaphore.h>
#include <sys/mman.h>

using namespace std;

#define SHM_NAME "/posix_shm_demo"
#define SHM_NAME_SEM "/posix_sem_demo"

char sharedMem[10];

int main()
{
    int fd;
    sem_t *sem;

    fd = shm_open(SHM_NAME, O_RDWR | O_CREAT, 0666);
    sem = sem_open(SHM_NAME_SEM, O_CREAT, 0666, 0);

    if (fd < 0 || sem == SEM_FAILED)
```

```
    {
        cout<<"shm_open or sem_open failed...";
        cout<<strerror(errno)<<endl;
        return -1;
    }

    ftruncate(fd, sizeof(sharedMem));

    char *memPtr;
    memPtr = (char *)mmap(NULL, sizeof(sharedMem), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);

    char msg[] = "yuki...";

    memmove(memPtr, msg, sizeof(msg));
    cout<<"process:"<<getpid()<<" send:"<<memPtr<<endl;

    sem_post(sem);
    sem_close(sem);

    return 0;
}

```

读进程代码：posix_read.cpp

```
//posix_read.cpp
#include <iostream>
#include <cstring>
#include <errno.h>

#include <unistd.h>
#include <fcntl.h>
#include <semaphore.h>
```

```cpp
#include <sys/mman.h>

using namespace std;

#define SHM_NAME "/posix_shm_demo"
#define SHM_NAME_SEM "/posix_sem_demo"

int main()
{
    int fd;
    sem_t *sem;

    fd = shm_open(SHM_NAME, O_RDWR, 0);
    sem = sem_open(SHM_NAME_SEM, 0);

    if (fd < 0 || sem == SEM_FAILED)
    {
        cout<<"shm_open or sem_open failed...";
        cout<<strerror(errno)<<endl;
        return -1;
    }

    struct stat fileStat;
    fstat(fd, &fileStat);

    char *memPtr;
    memPtr = (char *)mmap(NULL, fileStat.st_size, PROT_READ | PROT_WRITE, MAP_SHARED, f
d, 0);
    close(fd);

    sem_wait(sem);

    cout<<"process:"<<getpid()<<" recv:"<<memPtr<<endl;

    sem_close(sem);
```

```
43
44      return 0;
45  }
```

编译注意带上`lrt`和`lpthread`库：

```
➜  shm_demo git:(dev_hzg) ✗ ll
总用量 8.0K
-rwx------ 1 huzhigen huzhigen  913 10月 23 20:10 posix_read.cpp
-rwx------ 1 huzhigen huzhigen 1.1K 10月 23 20:10 posix_write.cpp
➜  shm_demo git:(dev_hzg) ✗ g++ posix_write.cpp -o posix_write -lrt -lpthread
➜  shm_demo git:(dev_hzg) ✗ g++ posix_read.cpp -o posix_read -lrt -lpthread
➜  shm_demo git:(dev_hzg) ✗
```

运行结果：

```
➜  shm_demo git:(dev_hzg) ✗ ./posix_write
process:10836 send:hello ryz'boys...
➜  shm_demo git:(dev_hzg) ✗ ./posix_read
process:10866 recv:hello ryz'boys...
➜  shm_demo git:(dev_hzg) ✗ ll /dev/shm
```

/dev/shm会新建对应名称文件，ipcs命令可以查看当前系统共享内存

```
→  shm_demo git:(dev_hzg) ✗ ll /dev/shm
总用量 80K
-rw-r--r-- 1 huzhigen huzhigen   50 10月 23 20:41 posix_shm_demo
-rw-r--r-- 1 huzhigen huzhigen 512M 10月 23 16:09 public_share_memory_666
-rw-r--r-- 1 huzhigen huzhigen   32 10月 23 16:26 sem.NamedMutex__666pub_pet_shm
-rw-r--r-- 1 huzhigen huzhigen   32 10月 23 16:09 sem.NamedMutex__666pub_role_base
-rw-r--r-- 1 huzhigen huzhigen   32 10月 23 16:24 sem.NamedMutex__666pub_role_face
-rw-r--r-- 1 huzhigen huzhigen   32 10月 23 16:24 sem.NamedMutex__666pub_role_fashion
-rw-r--r-- 1 huzhigen huzhigen   32 10月 23 16:24 sem.NamedMutex__666pub_role_plot
-rw-r--r-- 1 huzhigen huzhigen   32 10月 23 16:26 sem.NamedMutex__666pub_title_shm
-rw-r--r-- 1 huzhigen huzhigen   32 10月 23 16:09 sem.NamedMutex__666pub_update_queue
-rw-r--r-- 1 huzhigen huzhigen   32 10月 23 20:14 sem.posix_sem_demo
-rwxrwxrwx 1 huzhigen huzhigen    0 10月 18 17:48 v_test_1
drwxrwxrwx 3 huzhigen huzhigen   60 10月 15 14:26 wallstreet.55plBfmcn
→  shm_demo git:(dev_hzg) ✗ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 0          zabbix     600        365056     6          dest
0xffffffff 32769      huzhigen   0          4096       0
0x000015c6 65538      huzhigen   0          4096       0
0x00118d74 98307      huzhigen   0          4096       0
0x001193d3 163844     huzhigen   666        4096       0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x00000000 65536      zabbix     600        14

→  shm_demo git:(dev_hzg) ✗
```

System V IPC-SHM

内核实现剖析

系统调用mmap()通过映射一个普通文件实现共享内存。System V IPC则是通过映射特殊文件系统shm中的文件实现进程间的共享内存通信。也就是说，每个共享内存区域对应特殊文件系统shm中的一个文件。并且内核会通过**shmid_kernel**这个结构体来维护每一个共享内存对象。

系统V共享内存通过shmget获得或创建一个IPC共享内存区域，并返回相应的标识符。内核在保证shmget获得或创建一个共享内存区，初始化该共享内存区相应的shmid_kernel结构注同时，还将在特殊文件系统shm中，创建并打开一个同名文件，并在内存中建立起该文件的相应dentry及inode结构，新打开的文件不属于任何一个进程（任何进程都可以访问该共享内存区）。

每一个新创建的共享内存对象都用一个**shmid_kernel**数据结构来表达。系统中所有的**shmid_kernel**数据结构都保存在**shm_segs**向量表中，该向量表的每一个元素都是一个指向**shmid_kernel**数据结构的指针。

shm_segs向量表的定义如下：

```
struct shmid_kernel *shm_segs[SHMMNI];
```

SHMMNI为128，表示系统中最多可以有128个共享内存对象。当前可以通过修改`/proc/sys/kernel/shmmni`参数来修改SHMMNI的值，（还有一些其他的共享内存参数，都是在`/proc/sys/kernel/`这个目录下）

数据结构**shmid_kernel**的定义如下：

```
struct shmid_kernel
{
    struct shmid_ds u;
    struct file *  shm_file;
    unsigned long shm_npages;
    unsigned long *shm_pages;
    struct vm_area_struct *attaches;
};
```

其中：

shm_file存储了将被映射文件的地址。每个共享内存区对象都对应特殊文件系统shm中的一个文件，一般情况下，特殊文件系统shm中的文件是不能用read()、write()等方法访问的，当采取共享内存的方式把其中的文件映射到进程地址空间后，可直接采用访问内存的方式对其访问。

shm_pages代表该共享内存对象的所占据的内存页面数组，数组里面的每个元素当然是每个内存页面的起始地址。

shm_npages则是该共享内存对象占用内存页面的个数，以页为单位。这个数量当然涵盖了申请空间的最小整数倍。

shmid_ds是一个数据结构，它描述了这个共享内存区的认证信息，字节大小，最后一次粘附时间、分离时间、改变时间，创建该共享区域的进程，最后一次对它操作的进程，当前有多少个进程在使用它等信息。

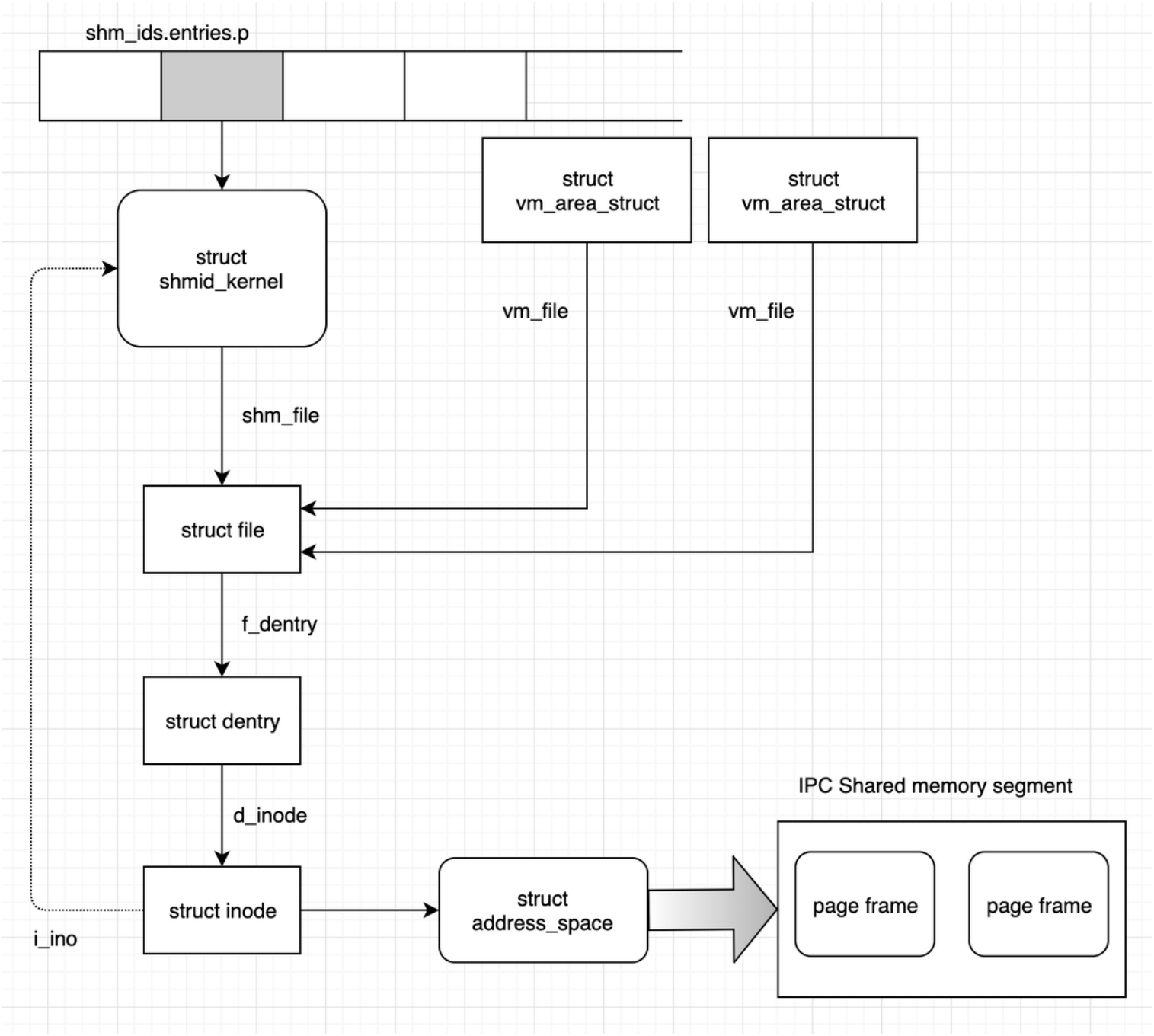
其定义如下：

```
struct shmid_ds{
    struct ipc_perm shm_perm;/* 操作权限*/
    int shm_segsz;                  /*段的大小（以字节为单位）*/
    time_t shm_atime;          /*最后一个进程附加到该段的时间*/
    time_t shm_dtime;          /*最后一个进程离开该段的时间*/
    time_t shm_ctime;          /*最后一个进程修改该段的时间*/
    unsigned short shm_cpid;   /*创建该段进程的pid*/
```

```
    unsigned short shm_lpid;   /*在该段上操作的最后1个进程的pid*/
    short shm_nattch;          /*当前附加到该段的进程的个数*/
    /*下面是私有的*/
    unsigned short shm_npages;  /*段的大小（以页为单位）*/
    unsigned long *shm_pages;   /*指向frames->SHMMAX的指针数组*/
    struct vm_area_struct *attaches; /*对共享段的描述*/
};
```

attaches描述被共享的物理内存对象所映射的各进程的虚拟内存区域。每一个希望共享这块内存的进程都必须通过系统调用将其关联（attach）到它的虚拟内存中。这一过程将为该进程创建了一个新的描述这块共享内存的vm_area_struct数据结构。创建时可以指定共享内存在它的虚拟地址空间的位置，也可以让Linux自己为它选择一块足够的空闲区域。

可以结合下图来理解数据结构



API接口：

看下面的例子

实例

写进程

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
```

```c
#include <string.h>
#include <stdio.h>

typedef struct {
    char name[4];
    int age;
} people;

int main(int argc, char** argv)
{
    int shm_id,i;
    key_t key = 9527;
    char temp;
    people *p_map;
    if(key==-1)
        perror("ftok error");

    shm_id = shmget(key,4096,IPC_CREAT|0666);
    if(shm_id==-1)
    {
        perror("shmget error");
        return 0;
    }
    p_map=(people*)shmat(shm_id,NULL,0);
    temp='a';
    for(i = 0;i<10;i++)
    {
        temp+=1;
        memcpy((*(p_map+i)).name,&temp,1);
        (*(p_map+i)).age=20+i;
    }
    if(shmdt(p_map)==-1)
        perror(" detach error ");

    return 0;
```

```
}
```

读进程

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

typedef struct {
    char name[4];
    int age;
} people;

int main(int argc, char** argv)
{
    int shm_id,i;
    key_t key = 9527;
    people *p_map;
    if(key == -1)
        perror("ftok error");

    shm_id = shmget(key,4096,IPC_CREAT|0666);
    if(shm_id == -1)
    {
        perror("shmget error");
        return 0;
    }
    p_map = (people*)shmat(shm_id,NULL,0);
    for(i = 0;i<10;i++)
    {
        printf( "name:%s\n",(*(p_map+i)).name );
        printf( "age %d\n",(*(p_map+i)).age );
    }
```

```
32      if(shmdt(p_map) == -1)
33          perror(" detach error ");
34
35      return 0;
36  }
```

结果：

```
huzhigen@genge in ~/work_space/codes/shm_demo
$ gcc system_v_wirte.c -o system_v_wirte
huzhigen@genge in ~/work_space/codes/shm_demo
$ gcc system_v_read.c -o system_v_read
huzhigen@genge in ~/work_space/codes/shm_demo
$ ./system_v_wirte
huzhigen@genge in ~/work_space/codes/shm_demo
$ ./system_v_read
name:b
age 20
name:c
age 21
name:d
age 22
name:e
age 23
name:f
age 24
name:g
age 25
name:h
age 26
name:i
age 27
name:j
age 28
name:k
age 29
huzhigen@genge in ~/work_space/codes/shm_demo
$
```

Posix与System V不同点：

> 二者主要差别在于Posix共享内存区大小可以通过ftruncate修改大小，而Sys对象大小在调用shmget创建时候固定下来了。

2. Window类系统

实现剖析

共享内存 (也叫内存映射文件) 主要是通过映射机制实现的 ,当调用 **CreateFileMapping** 创建命名的内存映射文件对象时 , Windows 即在物理内存申请一块指定大小的内存区域 , 返回文件映射对象的句柄 hMap ; 为了能够访问这块内存区域必须调用 **MapViewOfFile** 函数 , 促使 Windows 将此内存空间映射到进程的地址空间中

; 当在其他进程访问这块内存区域时 , 则必须使用 **OpenFileMapping** 函数取得对象句柄 hMap , 并调用 **MapViewOfFile** 函数得到此内存空间的一个映射 , 使用**UnmapViewOfFile**函数来解除映射，这样系统就把同一块内存区域映射到了不同进程的地址空间中 , 从而达到共享内存的目的 , 代码如下 。

实例

写进程代码

```
// WinShmWrite.cpp : Defines the entry point for the console application.//
#include "stdafx.h"
#include <windows.h>
#include <iostream>
using namespace std;
 #define BUF_SIZE 4096
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[]){
    // 定义共享数据
    char szBuffer[] = "Hello Shared Memory";

    // 创建共享文件句柄
    HANDLE hMapFile = CreateFileMapping(
            INVALID_HANDLE_VALUE,   // 物理文件句柄
            NULL,   // 默认安全级别
            PAGE_READWRITE,   // 可读可写
            0,   // 高位文件大小
            BUF_SIZE,   // 地位文件大小
            L"ShareMemory"   // 共享内存名称
    );

    // 映射缓存区视图 , 得到指向共享内存的指针
    LPVOID lpBase = MapViewOfFile(
            hMapFile,            // 共享内存的句柄
            FILE_MAP_ALL_ACCESS, // 可读写许可
            0,
            0,
            BUF_SIZE
    );

```

```
    // 将数据拷贝到共享内存
    strcpy((char*)lpBase, szBuffer);

    // 线程挂起等其他线程读取数据
    Sleep(20000);

    // 解除文件映射
    UnmapViewOfFile(lpBase);
    // 关闭内存映射文件对象句柄
    CloseHandle(hMapFile);
    std::cin.get();
    return 0;
}

```

读进程代码

```
// WinShmRead.cpp : Defines the entry point for the console application.//
#include "stdafx.h"
#include <iostream>
#include <windows.h>
using namespace std;
 #define BUF_SIZE 4096
int main(){
    // 打开共享的文件对象
    HANDLE hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, NULL, L"ShareMemory");
    if (hMapFile)
    {
            LPVOID lpBase = MapViewOfFile(hMapFile, FILE_MAP_ALL_ACCESS, 0, 0, 0);
            // 将共享内存数据拷贝出来
            char szBuffer[BUF_SIZE] = { 0 };
            strcpy(szBuffer, (char*)lpBase);
            printf("%s", szBuffer);

            // 解除文件映射
```

```
19              UnmapViewOfFile(lpBase);
20              // 关闭内存映射文件对象句柄
21              CloseHandle(hMapFile);
22      }
23      else
24      {
25              // 打开共享内存句柄失败
26              printf("OpenMapping Error");
27      }
28      std::cin.get();
29      return 0;
30  }
```

结果：

四、第三方支持

C++ STL支持

看过上面的例子后，是不是感觉接口还是很难用？原因之一就是申请或者映射内存返回的是内存指针，CRUD极不方便，无时不刻在关注内存偏移量，偏移错乱后，数据也就是乱掉了！！！

那怎么将STL和SHM结合呢？是的，就是自定义Allocation内存分配器。

这篇文章讲的挺好：http://blog.jqian.net/post/creating-stl-containers-in-shared-memory.html

不过工作量还是很大...

C++ Boost

boost应该是近乎标准的C++标准了，是每个学习C++的人都避不开的，因为实在是太强大，如果你发现std里面有没有你想要的API，尝试去boost看看。

服务器就是使用boost.interprocess部分提供的共享内存，极度舒适！

实例代码

```
//Demo_Boost unordered containers
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <boost/unordered_map.hpp>     //boost::unordered_map
#include <functional>                  //std::equal_to
#include <boost/functional/hash.hpp>   //boost::hash
int main()
{
    using namespace boost::interprocess;
    //Remove shared memory on construction and destruction
    struct shm_remove
    {
        shm_remove() { shared_memory_object::remove("MySharedMemory"); }
        ~shm_remove() { shared_memory_object::remove("MySharedMemory"); }
    } remover;

    //Create shared memory
    managed_shared_memory segment(create_only, "MySharedMemory", 65536);

    //Note that unordered_map<Key, MappedType>'s value_type is std::pair<const Key, MappedType>,
    //so the allocator must allocate that pair.
    typedef int    KeyType;
    typedef float  MappedType;
    typedef std::pair<const int, float> ValueType;
```

```
25
26      //Typedef the allocator
27      typedef allocator<ValueType, managed_shared_memory::segment_manager> ShmemAllocator;
28
29      //Alias an unordered_map of ints that uses the previous STL-like allocator.
30      typedef boost::unordered_map
31          < KeyType, MappedType
32          , boost::hash<KeyType>, std::equal_to<KeyType>
33          , ShmemAllocator>
34          MyHashMap;
35
36      //Construct a shared memory hash map.
37      //Note that the first parameter is the initial bucket count and
38      //after that, the hash function, the equality function and the allocator
39      MyHashMap *myhashmap = segment.construct<MyHashMap>("MyHashMap")  //object name
40          (3, boost::hash<int>(), std::equal_to<int>()                  //
41              , segment.get_allocator<ValueType>());                        //allocator in
    stance
42
43                                                                        //Insert data
    in the hash map
44      for (int i = 0; i < 100; ++i) {
45          myhashmap->insert(ValueType(i, (float)i));
46      }
47      return 0;
48  }
```

boost.interprocess 基于那种方式实现的呢?

Docker支持Shm吗

https://tonybai.com/2014/10/12/discussion-on-shared-mem-support-in-docker/
结论：Docker容器内本身是支持SHM的，可以将看作是物理机，重启后失效。但是如果使用mmap映射不同文件的方式，那么持久化，即使重启数据也还是在的。

五、不足之处

不能解决问题：

- 跨机器问题无能为力
- 容量大小不可动态变化
- 共享内存API没有提供对应的锁机制，只能配合系统信号量的来实现同步

六、参考文献

Unix环境高级编程 Advanced Programming in the UNIX Environment
深入理解LINUX内核 Understanding The Linux Kernel
https://www.ibm.com/developerworks/cn/linux/l-ipc/part5/index1.html
https://www.ibm.com/developerworks/cn/linux/l-ipc/part5/index2.html

附件shm_demo.zip