Linux进程间通信-共享内存

Linux进程间通信-共享内存

前言

本文主要说明在Linux环境上如何使用共享内存。阅读本文可以帮你解决以下问题：

1. 什么是共享内存和为什么要有共享内存？
2. 如何使用mmap进行共享内存？
3. 如何使用XSI共享内存？
4. 如何使用POSIX共享内存？
5. 如何使用hugepage共享内存以及共享内存的相关限制如何配置？
6. 共享内存都是如何实现的？

使用文件或管道进行进程间通信会有很多局限性，比如效率问题以及数据处理使用文件描述符而不如内存地址访问方便，于是多个进程以共享内存的方式进行通信就成了很自然要实现的IPC方案。Linux系统在编程上为我们准备了多种手段的共享内存方案。包括：

1. mmap内存共享映射。
2. XSI共享内存。
3. POSIX共享内存。

下面我们就来分别介绍一下这三种内存共享方式。

如果觉得本文有用，请刷二维码任意捐助：

mmap内存共享映射

mmap本来的是存储映射功能。它可以将一个文件映射到内存中，在程序里就可以直接使用内存地址对文件内容进行访问，这可以让程序对文件访问更方便。其相关调用API原型如下：

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

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

int munmap(void *addr, size_t length);
```

由于这个系统调用的特性可以用在很多场合，所以Linux系统用它实现了很多功能，并不仅局限于存储映射。在这主要介绍的就是用mmap进行多进程的内存共享功能。Linux产生子进程的系统调用是fork，根据fork的语义以及其实现，我们知道新产生的进程在内存地址空间上跟父进程是完全一致的。所以Linux的mmap实现了一种可以在父子进程之间共享内存地址的方式，其使用方法是：

1. 父进程将flags参数设置MAP_SHARED方式通过mmap申请一段内存。内存可以映射某个具体文件，也可以不映射具体文件（fd置为-1，flag设置为MAP_ANONYMOUS）。
2. 父进程调用fork产生子进程。之后在父子进程内都可以访问到mmap所返回的地址，就可以共享内存了。

我们写一个例子试一下，这次我们并发100个进程写共享内存来看看竞争条件racing的情况：

```
[zorro@zorrozou-pc0 sharemem]$ cat racing_mmap.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/file.h>
#include <wait.h>
#include <sys/mman.h>

#define COUNT 100
```

```
int do_child(int *count)
{
    int interval;

    /* critical section */
    interval = *count;
    interval++;
    usleep(1);
    *count = interval;
    /* critical section */

    exit(0);
}

int main()
{
    pid_t pid;
    int count;
    int *shm_p;

    shm_p = (int *)mmap(NULL, sizeof(int), PROT_WRITE|PROT_READ,
MAP_SHARED|MAP_ANONYMOUS, -1, 0);
    if (MAP_FAILED == shm_p) {
        perror("mmap()");
        exit(1);
    }

    *shm_p = 0;

    for (count=0;count<COUNT;count++) {
        pid = fork();
        if (pid < 0) {
            perror("fork()");
            exit(1);
        }

        if (pid == 0) {
            do_child(shm_p);
        }
    }

    for (count=0;count<COUNT;count++) {
        wait(NULL);
    }

    printf("shm_p: %d\n", *shm_p);
```

```
    munmap(shm_p, sizeof(int));
    exit(0);
}
```

这个例子中，我们在子进程中为了延长临界区（critical section）处理的时间，使用了一个中间变量进行数值交换，并且还使用了usleep加强了一下racing的效果，最后执行结果：

```
[zorro@zorrozou-pc0 sharemem]$ ./racing_mmap
shm_p: 20
[zorro@zorrozou-pc0 sharemem]$ ./racing_mmap
shm_p: 17
[zorro@zorrozou-pc0 sharemem]$ ./racing_mmap
shm_p: 14
[zorro@zorrozou-pc0 sharemem]$ ./racing_mmap
shm_p: 15
```

这段共享内存的使用是有竞争条件存在的，从文件锁的例子我们知道，进程间通信绝不仅仅是通信这么简单，还需要处理类似这样的临界区代码。在这里，我们也可以使用文件锁进行处理，但是共享内存使用文件锁未免显得太不协调了。除了不方便以及效率低下以外，文件锁还不能够进行更高级的进程控制。所以，我们在此需要引入更高级的进程同步控制原语来实现相关功能，这就是信号量（semaphore）的作用。我们会在后续章节中集中讲解信号量的使用，在此只需了解使用mmap共享内存的方法。

我们有必要了解一下mmap的内存占用情况，以便知道使用它的成本。我们申请一段比较大的共享内存进行使用，并察看内存占用情况。测试代码：

```
[zorro@zorrozou-pc0 sharemem]$ cat mmap_mem.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/file.h>
#include <wait.h>
#include <sys/mman.h>
```

```
#define COUNT 100
#define MEMSIZE 1024*1024*1023*2

int main()
{
    pid_t pid;
    int count;
    void *shm_p;

    shm_p = mmap(NULL, MEMSIZE, PROT_WRITE|PROT_READ,
MAP_SHARED|MAP_ANONYMOUS, -1, 0);
    if (MAP_FAILED == shm_p) {
        perror("mmap()");
        exit(1);
    }

    bzero(shm_p, MEMSIZE);

    sleep(3000);

    munmap(shm_p, MEMSIZE);
    exit(0);
}
```

我们申请一段大概2G的共享内存，并置0。然后在执行前后分别看内存用量，看什么部分有变化：

```
 [zorro@zorrozou-pc0 sharemem]$ free -g
              total        used        free      shared  buff/cache
available
Mem:             15           2           2           0          10
11
Swap:            31           0          31
[zorro@zorrozou-pc0 sharemem]$ ./mmap_mem &
[1] 32036
[zorro@zorrozou-pc0 sharemem]$ free -g
              total        used        free      shared  buff/cache
available
Mem:             15           2           0           2          12
9
Swap:            31           0          31
```

可以看到，这部分内存的使用被记录到了shared和buff/cache中。当然这个结果在不同版本的Linux上可能是不一样的，比如在Centos 6的环境中mmap的共享内存只会记录到buff/cache中。除了占用空间的问题，还应该注意，mmap方式的共享内存只能在通过fork产生的父子进程间通信，因为除此之外的其它进程无法得到共享内存段的地址。

XSI共享内存

为了满足多个无关进程共享内存的需求，Linux提供了更具通用性的共享内存手段，XSI共享内存就是这样一种实现。XSI是X/Open组织对UNIX定义的一套接口标准(X/Open System Interface)。由于UNIX系统的历史悠久，在不同时间点的不同厂商和标准化组织定义过一些列标准，而目前比较通用的标准实际上是POSIX。我们还会经常遇到的标准还包括SUS（Single UNIX Specification）标准，它们大概的关系是，SUS是POSIX标准的超集，定义了部分额外附加的接口，这些接口扩展了基本的POSIX规范。相应的系统接口全集被称为XSI标准，除此之外XSI还定义了实现必须支持的POSIX的哪些可选部分才能认为是遵循XSI的。它们包括文件同步，存储映射文件，存储保护及线程接口。只有遵循XSI标准的实现才能称为UNIX操作系统。

XSI共享内存在Linux底层的实现实际上跟mmap没有什么本质不同，只是在使用方法上有所区别。其使用的相关方法为：

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

int shmget(key_t key, size_t size, int shmflg);

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

#include <sys/types.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);

int shmdt(const void *shmaddr);
```

我们首先要来理解的是key这一个参数。想象一下我们现在需要解决的问题：“在一个操作系统内，如何让两个不相关（没有父子关系）的进程可以共享一个内存段？”系统中是否有现成的解决方案呢？当然有，就是文件。我们知道，文件的设计就可以让无关的进程可以进行数据交换。文件采用路径和文件名作为系统全局的一个标识符，但是每个进程打开这个文件之后，在进程内部都有一个“文件描述符”去指向文件。此时进程通过fork打开的子进程可以继承父进程的文件描述符，但是无关进程依然可以通过系统全局的文件名用open系统调用再次打开同一个文件，以便进行进程间通信。

实际上对于XSI的共享内存，其key的作用就类似文件的文件名，shmget返回的int类型的shmid就类似文件描述符，注意只是“类似”，而并非是同样的实现。这意味着，我们在进程中不能用select、poll、epoll这样的方法去控制一个XSI共享内存，因为它并不是“文件描述符”。对于一个XSI的共享内存，其key是系统全局唯一的，这就方便其他进程使用同样的key，打开同样一段共享内存，以便进行进程间通信。而使用fork产生的子进程，则可以直接通过shmid访问到相关共享内存段。这就是key的本质：系统中对XSI共享内存的全局唯一表示符。

如果觉得本文有用，请刷二维码任意捐助：

明白了这个本质之后，我们再来看看这个key应该如何产生。相关方法为：

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

key_t ftok(const char *pathname, int proj_id);
```

一个key是通过ftok函数，使用一个pathname和一个proj_jd产生的。就是说，在一个可能会使用共享内存的项目组中，大家可以约定一个文件名和一个项目的proj_id，来在同一个系统中确定一段共享内存的key。ftok并不会去创建文件，所以必须指定一个存在并且进程可以访问的pathname路径。这里还要指出的一点是，ftok实际上并不是根据文件的文件路径和文件名（pathname）产生key的，在实现上，它使用的是指定文件的inode编号和文件所在设备的设备编号。所以，不要以为你是用了不同的文件名就一定会得到不同的key，因为不同的文件名是可以指向相同inode编号的文件的（硬连接）。也不要认为你是用了相同的文件名就一定可以得到

相同的key，在一个系统上，同一个文件名会被删除重建的几率是很大的，这种行为很有可能导致文件的inode变化。所以一个ftok的执行会隐含stat系统调用也就不难理解了。

最后大家还应该明白，key作为全局唯一标识不仅仅体现在XSI的共享内存中，XSI标准的其他进程间通信机制（信号量数组和消息队列）也使用这一命名方式。这部分内容在《UNIX环境高级编程》一书中已经有了很详尽的讲解，本文不在赘述。我们还是只来看一下它使用的例子，我们用XSI的共享内存来替换刚才的mmap：

```
 [zorro@zorrozou-pc0 sharemem]$ cat racing_xsi_shm.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/file.h>
#include <wait.h>
#include <sys/mman.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>

#define COUNT 100
#define PATHNAME "/etc/passwd"

int do_child(int proj_id)
{
    int interval;
    int *shm_p, shm_id;
    key_t shm_key;
    /* 使用ftok产生shmkey */
    if ((shm_key = ftok(PATHNAME, proj_id)) == -1) {
        perror("ftok()");
        exit(1);
    }
    /* 在子进程中使用shmget取到已经在父进程中创建好的共享内存id，注意shmget
的第三个参数的使用。 */
    shm_id = shmget(shm_key, sizeof(int), 0);
    if (shm_id < 0) {
        perror("shmget()");
        exit(1);
    }
```

```
    /* 使用shmat将相关共享内存段映射到本进程的内存地址。 */

    shm_p = (int *)shmat(shm_id, NULL, 0);
    if ((void *)shm_p == (void *)-1) {
        perror("shmat()");
        exit(1);
    }

    /* critical section */
    interval = *shm_p;
    interval++;
    usleep(1);
    *shm_p = interval;
    /* critical section */

    /* 使用shmdt解除本进程内对共享内存的地址映射，本操作不会删除共享内存。
*/
    if (shmdt(shm_p) < 0) {
        perror("shmdt()");
        exit(1);
    }

    exit(0);
}

int main()
{
    pid_t pid;
    int count;
    int *shm_p;
    int shm_id, proj_id;
    key_t shm_key;

    proj_id = 1234;

    /* 使用约定好的文件路径和proj_id产生shm_key。 */
    if ((shm_key = ftok(PATHNAME, proj_id)) == -1) {
        perror("ftok()");
        exit(1);
    }

    /* 使用shm_key创建一个共享内存，如果系统中已经存在此共享内存则报错退出，
创建出来的共享内存权限为0600。 */
    shm_id = shmget(shm_key, sizeof(int), IPC_CREAT|IPC_EXCL|0600);
    if (shm_id < 0) {
```

```
    perror("shmget()");
    exit(1);
}

/* 将创建好的共享内存映射进父进程的地址以便访问。 */
shm_p = (int *)shmat(shm_id, NULL, 0);
if ((void *)shm_p == (void *)-1) {
    perror("shmat()");
    exit(1);
}

/* 共享内存赋值为0。 */
*shm_p = 0;

/*  打开100个子进程并发读写共享内存。 */
for (count=0;count<COUNT;count++) {
    pid = fork();
    if (pid < 0) {
        perror("fork()");
        exit(1);
    }

    if (pid == 0) {
        do_child(proj_id);
    }
}

/* 等待所有子进程执行完毕。 */
for (count=0;count<COUNT;count++) {
    wait(NULL);
}

/* 显示当前共享内存的值。 */
printf("shm_p: %d\n", *shm_p);


/* 解除共享内存地质映射。 */
if (shmdt(shm_p) < 0) {
    perror("shmdt()");
    exit(1);
}

/* 删除共享内存。 */
if (shmctl(shm_id, IPC_RMID, NULL) < 0) {
    perror("shmctl()");
    exit(1);
```

```
    }

    exit(0);
}
```

XSI共享内存跟mmap在实现上并没有本质区别。而之所以引入key和shmid的概念，也主要是为了在非父子关系的进程之间可以共享内存。根据上面的例子可以看到，使用shmget可以根据key创建共享内存，并返回一个shmid。它的第二个参数size用来指定共享内存段的长度，第三个参数指定创建的标志，可以支持的标志为：IPC_CREAT、IPC_EXCL。从Linux 2.6之后，还引入了支持大页的共享内存，标志为：SHM_HUGETLB、SHM_HUGE_2MB等参数。shmget除了可以创建一个新的共享内存以外，还可以访问一个已经存在的共享内存，此时可以将shmflg置为0，不加任何标识打开。

在某些情况下，我们也可以不用通过一个key来生成共享内存。此时可以在key的参数所在位置填：IPC_PRIVATE，这样内核会在保证不产生冲突的共享内存段id的情况下新建一段共享内存。当然，这样调用则必然意味着是新创建，而不是打开已有得共享内存，所以flag位一定是IPC_CREAT。此时产生的共享内存只有一个shmid，而没有key，所以可以通过fork的方式将id传给子进程。

当获得shmid之后，就可以使用shmat来进行地址映射。shmat之后，通过访问返回的当前进程的虚拟地址就可以访问到共享内存段了。当然，在使用之后要记得使用shmdt解除映射，否则对于长期运行的程序可能造成虚拟内存地址泄漏，导致没有可用地址可用。shmdt并不能删除共享内存段，而只是解除共享内存和进程虚拟地址的映射，只要shmid对应的共享内存还存在，就仍然可以继续使用shmat映射使用。想要删除一个共享内存需要使用shmctl的IPC_RMID指令处理。也可以在命令行中使用ipcrm删除指定的共享内存id或key。

> *共享内存由于其特性，与进程中的其他内存段在使用习惯上有些不同。一般进程对栈空间分配可以自动回收，而堆空间通过malloc申请，free回收。这些内存在回收之后就可以认为是不存在了。但是共享内存不同，用shmdt之后，实际上其占用的内存还在，并仍然可以使用shmat映射使用。如果不是用shmctl或ipcrm命令删除的话，那么它将一直保留直到系统被关闭。对于刚接触共享内存的程序员来说这可能需要适应一下。实际上共享内存的生存周期根文件更像：进程对文件描述符执行close并不能删除文件，而只是关闭了本进程对文件的操作接口，这就像shmdt的作用。而真正删除文件要用*

> *unlink，活着使用rm命令，这就像是共享内存的shmctl的IPC_RMID和ipcrm命令。当然，文件如果不删除，下次重启依旧还在，因为它放在硬盘上，而共享内存下次重启就没了，因为它毕竟还是内存。*

在这里，请大家理解关于为什么要使用key，和相关共享内存id的概念。后续我们还将看到，除了共享内存外，XSI的信号量、消息队列也都是通过这种方式进行相关资源标识的。

除了可以删除一个共享内存以外，shmctl还可以查看、修改共享内存的相关属性。这些属性的细节大家可以man 2 shmctl查看细节帮助。在系统中还可以使用ipcs -m命令查看系统中所有共享内存的的信息，以及ipcrm指定删除共享内存。

这个例子最后执行如下：

```
[zorro@zorrozou-pc0 sharemem]$ ./racing_xsi_shm
shm_p: 27
[zorro@zorrozou-pc0 sharemem]$ ./racing_xsi_shm
shm_p: 22
[zorro@zorrozou-pc0 sharemem]$ ./racing_xsi_shm
shm_p: 21
[zorro@zorrozou-pc0 sharemem]$ ./racing_xsi_shm
shm_p: 20
```

到目前为止，我们仍然没解决racing的问题，所以得到的结果仍然是不确定的，我们会在讲解信号量的时候引入锁解决这个问题，当然也可以用文件锁。我们下面再修改刚才的mmap_mem.c程序，换做shm方式再来看看内存的使用情况，代码如下：

```
[zorro@zorrozou-pc0 sharemem]$ cat xsi_shm_mem.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/file.h>
#include <wait.h>
#include <sys/mman.h>
```

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

#define COUNT 100
#define MEMSIZE 1024*1024*1023*2

int main()
{
    pid_t pid;
    int count, shm_id;
    void *shm_p;

    shm_id = shmget(IPC_PRIVATE, MEMSIZE, 0600);
    if (shm_id < 0) {
        perror("shmget()");
        exit(1);
    }


    shm_p = shmat(shm_id, NULL, 0);
    if ((void *)shm_p == (void *)-1) {
        perror("shmat()");
        exit(1);
    }

    bzero(shm_p, MEMSIZE);

    sleep(3000);

    if (shmdt(shm_p) < 0) {
        perror("shmdt()");
        exit(1);
    }

    if (shmctl(shm_id, IPC_RMID, NULL) < 0) {
        perror("shmctl()");
        exit(1);
    }

    exit(0);
}
```

我们在这段代码中使用了IPC_PRIVATE方式共享内存，这是与前面程序不一样的地方。执行结果为：

```
 [zorro@zorrozou-pc0 sharemem]$ free -g
              total        used        free      shared  buff/cache
available
Mem:             15           2           2           0          10
11
Swap:            31           0          31
[zorro@zorrozou-pc0 sharemem]$ ./xsi_shm_mem &
[1] 4539
[zorro@zorrozou-pc0 sharemem]$ free -g
              total        used        free      shared  buff/cache
available
Mem:             15           2           0           2          12
9
Swap:            31           0          31
```

跟mmap的共享内存一样，XSI的共享内存在free现实中也会占用shared和buff/cache的消耗。实际上，在内核底层实现上，两种内存共享都是使用的tmpfs方式实现的，所以它们实际上的内存使用都是一致的。

如果觉得本文有用，请刷二维码任意捐助：

对于Linux系统来说，使用XSI共享内存的时候可以通过shmget系统调用的shmflg参数来申请大页内存（huge pages），当然这样做将使进程的平台移植性变差。相关的参数包括：

SHM_HUGETLB (since Linux 2.6)

SHM_HUGE_2MB, SHM_HUGE_1GB (since Linux 3.8)

使用大页内存的好处是提高内核对内存管理的处理效率，这主要是因为在相同内存大小的情况下，使用大页内存（2M一页）将比使用一般内存页（4k一页）的内存页管理的数量大大减少，从而减少了内存页表项的缓存压力和CPU cache缓存内存地质映射的压力，提高了寻址能力和内存管理效率。大页内存还有其他一些使用时需要注意的地方：

1. 大页内存不能交换（SWAP）.
2. 使用不当时可能造成更大的内存泄漏。

我们继续使用上面的程序修改改为使用大页内存来做一下测试，大页内存需要使用root权限，代码跟上面程序一样，只需要修改一下shmget的参数，如下：

```
[root@zorrozou-pc0 sharemem]# cat xsi_shm_mem_huge.c
......
    shm_id = shmget(IPC_PRIVATE, MEMSIZE, SHM_HUGETLB|0600);
......
```

其余代码都不变。我们申请的内存大约不到2G，所以需要在系统内先给我们预留2G以上的大页内存：

```
[root@zorrozou-pc0 sharemem]# echo 2048 > /proc/sys/vm/nr_hugepages
[root@zorrozou-pc0 sharemem]# cat /proc/meminfo |grep -i huge
AnonHugePages:    841728 kB
HugePages_Total:    2020
HugePages_Free:     2020
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
```

2048是页数，每页2M，所以这里预留了几乎4G的内存空间给大页。之后我么好还需要确保共享内存的限制不会使我们申请失败：

```
[root@zorrozou-pc0 sharemem]# echo 2147483648 > /proc/sys/kernel/shmmax
[root@zorrozou-pc0 sharemem]# echo 33554432 > /proc/sys/kernel/shmall
```

之后编译执行相关命令：

```
[root@zorrozou-pc0 sharemem]# echo 1 > /proc/sys/vm/drop_caches
[root@zorrozou-pc0 sharemem]# free -g
              total        used        free      shared  buff/cache
available
```

```
Mem:             15           6           6           0           2
7
Swap:            31           0          31
[root@zorrozou-pc0 sharemem]# ./xsi_shm_mem_huge &
[1] 5508
[root@zorrozou-pc0 sharemem]# free -g
              total        used        free      shared  buff/cache
available
Mem:             15           6           6           0           2
7
Swap:            31           0          31
[root@zorrozou-pc0 sharemem]# cat /proc/meminfo |grep -i huge
AnonHugePages:    841728 kB
HugePages_Total:    2020
HugePages_Free:      997
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
```

大家可以根据这个程序的输出看到，当前系统环境（archlinux kernel 4.5）再使用大页内存之后，free命令是看不见其内存统计的。同样的设置在Centos 7环境下也是相同的显示。这就是说，如果使用大页内存作为共享内存使用，将在free中看不到相关内存统计，这估计是free命令目前暂时没将大页内存统计进内存使用所导致，暂时大家只能通过/proc/meminfo文件中的相关统计看到大页内存的使用信息。

XSI共享内存的系统相关限制如下：

/proc/sys/kernel/shmall：限制系统用在共享内存上的内存总页数。注意是页数，单位为4k。

/proc/sys/kernel/shmmax：限制一个共享内存段的最大长度，字节为单位。

/proc/sys/kernel/shmmni：限制整个系统可创建的最大的共享内存段个数。

XSI共享内存是历史比较悠久，也比较经典的共享内存手段。它几乎代表了共享内存的默认定义，当我们说有共享内存的时候，一般意味着使用了XSI的共享内存。但是这种共享内存也存在一切缺点，最受病垢的地方莫过于他提供的key＋projid的命名方式不够UNIX，没有遵循一切皆文件的设计理念。当然这个设计理念在一般的应用场景下并不是什么必须要遵守的理念，但是如果共享内存可以用文件描述符的方式提供给程序访问，毫无疑问可以在Linux上跟

select、poll、epoll这样的IO异步事件驱动机制配合使用，做到一些更高级的功能。于是，遵循一切皆文件理念的POSIX标准的进程间通信机制应运而生。

POSIX共享内存

POSIX共享内存实际上毫无新意，它本质上就是mmap对文件的共享方式映射，只不过映射的是tmpfs文件系统上的文件。

什么是tmpfs？Linux提供一种“临时”文件系统叫做tmpfs，它可以将内存的一部分空间拿来当做文件系统使用，使内存空间可以当做目录文件来用。现在绝大多数Linux系统都有一个叫做/dev/shm的tmpfs目录，就是这样一种存在。具体使用方法，大家可以参考我的另一篇文章《Linux内存中的Cache真的能被回收么？》。

Linux提供的POSIX共享内存，实际上就是在/dev/shm下创建一个文件，并将其mmap之后映射其内存地址即可。我们通过它给定的一套参数就能猜到它的主要函数shm_open无非就是open系统调用的一个封装。大家可以通过man shm_overview来查看相关操作的方法。使用代码如下：

```
[root@zorrozou-pc0 sharemem]# cat racing_posix_shm.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/file.h>
#include <wait.h>
#include <sys/mman.h>

#define COUNT 100
#define SHMPATH "shm"

int do_child(char * shmpath)
{
    int interval, shmfd, ret;
    int *shm_p;
    /* 使用shm_open访问一个已经创建的POSIX共享内存 */
```

```
    shmfd = shm_open(shmpath, O_RDWR, 0600);
    if (shmfd < 0) {
        perror("shm_open()");
        exit(1);
    }

    /* 使用mmap将对应的tmpfs文件映射到本进程内存 */
    shm_p = (int *)mmap(NULL, sizeof(int), PROT_WRITE|PROT_READ, MAP_SHARED,
shmfd, 0);
    if (MAP_FAILED == shm_p) {
        perror("mmap()");
        exit(1);
    }
    /* critical section */
    interval = *shm_p;
    interval++;
    usleep(1);
    *shm_p = interval;
    /* critical section */
    munmap(shm_p, sizeof(int));
    close(shmfd);

    exit(0);
}

int main()
{
    pid_t pid;
    int count, shmfd, ret;
    int *shm_p;

    /* 创建一个POSIX共享内存 */
    shmfd = shm_open(SHMPATH, O_RDWR|O_CREAT|O_TRUNC, 0600);
    if (shmfd < 0) {
        perror("shm_open()");
        exit(1);
    }

    /* 使用ftruncate设置共享内存段大小 */
    ret = ftruncate(shmfd, sizeof(int));
    if (ret < 0) {
        perror("ftruncate()");
        exit(1);
    }

    /* 使用mmap将对应的tmpfs文件映射到本进程内存 */
```

```
    shm_p = (int *)mmap(NULL, sizeof(int), PROT_WRITE|PROT_READ, MAP_SHARED,
shmfd, 0);
    if (MAP_FAILED == shm_p) {
        perror("mmap()");
        exit(1);
    }

    *shm_p = 0;

    for (count=0;count<COUNT;count++) {
        pid = fork();
        if (pid < 0) {
            perror("fork()");
            exit(1);
        }

        if (pid == 0) {
            do_child(SHMPATH);
        }
    }

    for (count=0;count<COUNT;count++) {
        wait(NULL);
    }

    printf("shm_p: %d\n", *shm_p);
    munmap(shm_p, sizeof(int));
    close(shmfd);
    //sleep(3000);
    shm_unlink(SHMPATH);
    exit(0);
}
```

编译执行这个程序需要指定一个额外**rt**的库，可以使用如下命令进行编译：

```
 [root@zorrozou-pc0 sharemem]# gcc -o racing_posix_shm -lrt
racing_posix_shm.c
```

对于这个程序，我们需要解释以下几点：

1. shm_open的SHMPATH参数是一个路径，这个路径默认放在系统的/dev/shm目录下。这是shm_open已经封装好的，保证了文件一定会使用tmpfs。
2. shm_open实际上就是open系统调用的封装。我们当然完全可以使用open的方式模拟这个方法。
3. 使用ftruncate方法来设置“共享内存”的大小。其实就是更改文件的长度。
4. 要以共享方式做mmap映射，并且指定文件描述符为shmfd。
5. shm_unlink实际上就是unlink系统调用的封装。如果不做unlink操作，那么文件会一直存在于/dev/shm目录下，以供其它进程使用。
6. 关闭共享内存描述符直接使用close。

以上就是POSIX共享内存。其本质上就是个tmpfs文件。那么从这个角度说，mmap匿名共享内存、XSI共享内存和POSIX共享内存在内核实现本质上其实都是tmpfs。如果我们去查看POSIX共享内存的free空间占用的话，结果将跟mmap和XSI共享内存一样占用shared和buff/cache，所以我们就不再做这个测试了。这部分内容大家也可以参考《Linux内存中的Cache真的能被回收么？》。

根据以上例子，我们整理一下POSIX共享内存的使用相关方法：

```
 #include <sys/mman.h>
#include <sys/stat.h>        /* For mode constants */
#include <fcntl.h>           /* For O_* constants */

int shm_open(const char *name, int oflag, mode_t mode);

int shm_unlink(const char *name);
```

使用shm_open可以创建或者访问一个已经创建的共享内存。上面说过，实际上POSIX共享内存就是在/dev/shm目录中的的一个tmpfs格式的文件，所以shm_open无非就是open系统调用的封装，所以起函数使用的参数几乎一样。其返回的也是一个标准的我呢间描述符。

shm_unlink也一样是unlink调用的封装，用来删除文件名和文件的映射关系。在这就能看出POSIX共享内存和XSI的区别了，一个是使用文件名作为全局标识，另一个是使用key。

映射共享内存地址使用mmap，解除映射使用munmap。使用ftruncate设置共享内存大小，实际上就是对tmpfs的文件进行指定长度的截断。使用fchmod、fchown、fstat等系统调用修改和查看相关共享内存的属性。close调用关闭共享内存的描述符。实际上，这都是标准的文件操作。

最后

希望这些内容对大家进一步深入了共享内存有帮助。如果有相关问题，可以在我的微博、微信或者博客上联系我。

如果觉得本文有用，请刷二维码任意捐助：

大家好，我是Zorro！

如果你喜欢本文，欢迎在微博上搜索“**orroz**”关注我，地址是：http://weibo.com/orroz

大家也可以在微信上搜索：**Linux系统技术** 关注我的公众号。

我的所有文章都会沉淀在我的个人博客上，地址是：http://liwei.life。

欢迎使用以上各种方式一起探讨学习，共同进步。

公众号二维码：

共享此文章：

相关:

Cgroup - Linux内存资源管理

一月 22, 2016
在“Linux”中

Linux内存中的Cache真的能被回收么？

四月 26, 2016
在“Linux”中

Linux的内存回收和交换

六月 27, 2016
含有 1 条评论

zorro / 八月 8, 2016 / Uncategorized