一文搞定 | Linux共享内存原理

HappenLee Linux学习 今天

> **来自：云社区，作者：HappenLee**
> **链接：https://cloud.tencent.com/developer/article/1396351**

在Linux系统中，每个进程都有独立的虚拟内存空间，也就是说不同的进程访问同一段虚拟内存地址所得到的数据是不一样的，这是因为不同进程相同的虚拟内存地址会映射到不同的物理内存地址上。

但有时候为了让不同进程之间进行通信，需要让不同进程共享相同的物理内存，Linux通过 `共享内存` 来实现这个功能。下面先来介绍一下Linux系统的共享内存的使用。

共享内存使用

1. 获取共享内存

要使用共享内存，首先需要使用 `shmget()` 函数获取共享内存，`shmget()` 函数的原型如下：

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

- 参数 `key` 一般由 `ftok()` 函数生成，用于标识系统的唯一IPC资源。
- 参数 `size` 指定创建的共享内存大小。

- 参数 `shmflg` 指定 `shmget()` 函数的动作，比如传入 `IPC_CREAT` 表示要创建新的共享内存。

函数调用成功时返回一个新建或已经存在的的共享内存标识符，取决于shmflg的参数。失败返回-1，并设置错误码。

2. 关联共享内存

`shmget()` 函数返回的是一个标识符，而不是可用的内存地址，所以还需要调用 `shmat()` 函数把共享内存关联到某个虚拟内存地址上。`shmat()` 函数的原型如下：

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

- 参数 `shmid` 是 `shmget()` 函数返回的标识符。
- 参数 `shmaddr` 是要关联的虚拟内存地址，如果传入0，表示由系统自动选择合适的虚拟内存地址。
- 参数 `shmflg` 若指定了 `SHM_RDONLY` 位，则以只读方式连接此段，否则以读写方式连接此段。

函数调用成功返回一个可用的指针（虚拟内存地址），出错返回-1。

3. 取消关联共享内存

当一个进程不需要共享内存的时候，就需要取消共享内存与虚拟内存地址的关联。取消关联共享内存通过 `shmdt()` 函数实现，原型如下：

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

- 参数 `shmaddr` 是要取消关联的虚拟内存地址，也就是 `shmat()` 函数返回的值。

函数调用成功返回0，出错返回-1。

共享内存使用例子

下面通过一个例子来介绍一下共享内存的使用方法。在这个例子中，有两个进程，分别为 `进程A` 和 `进程B`，`进程A` 创建一块共享内存，然后写入数据，`进程B` 获取这块共享内存并且读取其内容。

进程A

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

#define SHM_PATH "/tmp/shm"
#define SHM_SIZE 128

int main(int argc, char *argv[])
{
    int shmid;
    char *addr;
    key_t key = ftok(SHM_PATH, 0x6666);

    shmid = shmget(key, SHM_SIZE, IPC_CREAT|IPC_EXCL|0666);
```

```
    if (shmid < 0) {
        printf("failed to create share memory\n");
        return -1;
    }

    addr = shmat(shmid, NULL, 0);
    if (addr <= 0) {
        printf("failed to map share memory\n");
        return -1;
    }

    sprintf(addr, "%s", "Hello World\n");

    return 0;
}
```

进程B

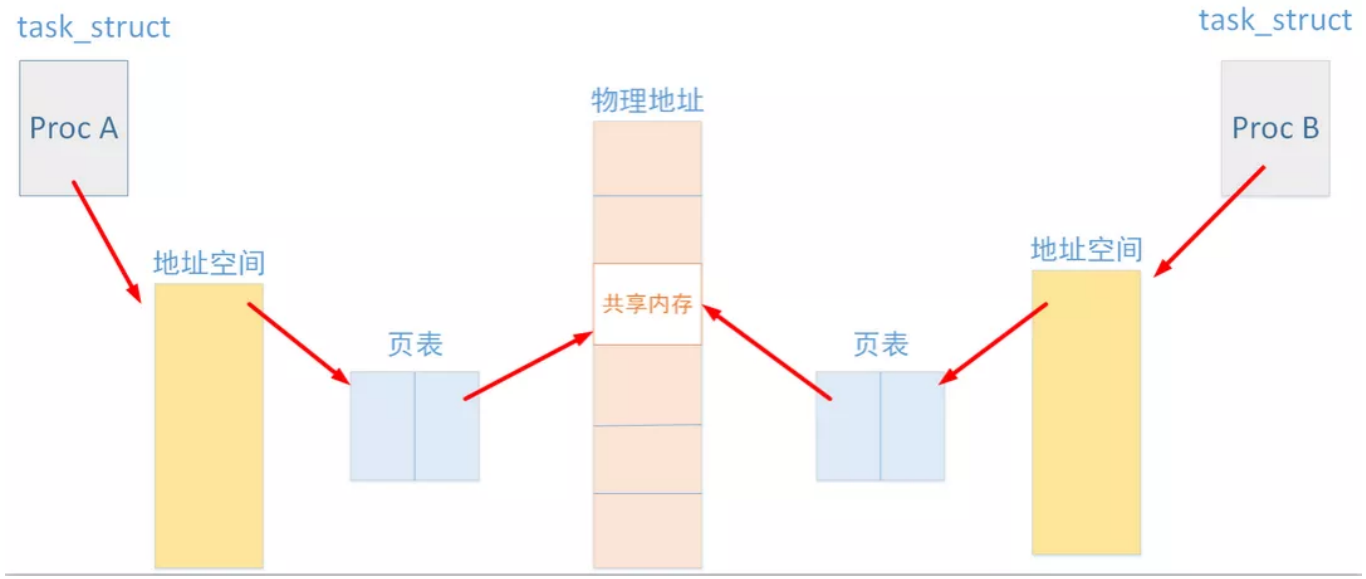
```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_PATH "/tmp/shm"
#define SHM_SIZE 128
```

```c
int main(int argc, char *argv[])
{
    int shmid;
    char *addr;
    key_t key = ftok(SHM_PATH, 0x6666);

    char buf[128];

    shmid = shmget(key, SHM_SIZE, IPC_CREAT);
    if (shmid < 0) {
        printf("failed to get share memory\n");
        return -1;
    }

    addr = shmat(shmid, NULL, 0);
    if (addr <= 0) {
        printf("failed to map share memory\n");
        return -1;
    }

    strcpy(buf, addr, 128);
    printf("%s", buf);

    return 0;
}
```

测试时先运行进程A，然后再运行进程B，可以看到进程B会打印出 “Hello World”，说明共享内存已经创建成功并且读取。

共享内存实现原理

我们先通过一幅图来了解一下共享内存的大概原理，如下图：



通过上图可知，共享内存是通过将不同进程的虚拟内存地址映射到相同的物理内存地址来实现的，下面将会介绍Linux的实现方式。

在Linux内核中，每个共享内存都由一个名为 `struct shmid_kernel` 的结构体来管理，而且Linux限制了系统最大能创建的共享内存为128个。通过类型为 `struct shmid_kernel` 结构的数组来管理，如下：

```
struct shmid_ds {
 struct ipc_perm  shm_perm; /* operation perms */
 int   shm_segsz; /* size of segment (bytes) */
 __kernel_time_t  shm_atime; /* last attach time */
 __kernel_time_t  shm_dtime; /* last detach time */
 __kernel_time_t  shm_ctime; /* last change time */
 __kernel_ipc_pid_t shm_cpid; /* pid of creator */
```

```
 __kernel_ipc_pid_t shm_lpid; /* pid of last operator */
 unsigned short  shm_nattch; /* no. of current attaches */
 unsigned short   shm_unused; /* compatibility */
 void    *shm_unused2; /* ditto - used by DIPC */
 void   *shm_unused3; /* unused */
};

struct shmid_kernel
{
 struct shmid_ds  u;
 /* the following are private */
 unsigned long  shm_npages; /* size of segment (pages) */
 pte_t   *shm_pages; /* array of ptrs to frames -> SHMMAX */
 struct vm_area_struct *attaches; /* descriptors for attaches */
};

static struct shmid_kernel *shm_segs[SHMMNI]; // SHMMNI等于128
```

从注释可以知道 `struct shmid_kernel` 结构体各个字段的作用，比如 `shm_npages` 字段表示共享内存使用了多少个内存页。而 `shm_pages` 字段指向了共享内存映射的虚拟内存页表项数组等。

另外 `struct shmid_ds` 结构体用于管理共享内存的信息，而 `shm_segs数组` 用于管理系统中所有的共享内存。

shmget() 函数实现

通过前面的例子可知，要使用共享内存，首先需要调用 `shmget()` 函数来创建或者获取一块共享内存。`shmget()` 函数的实现如下：

```
asmlinkage long sys_shmget (key_t key, int size, int shmflg)
{
 struct shmid_kernel *shp;
 int err, id = 0;

 down(&current->mm->mmap_sem);
 spin_lock(&shm_lock);
 if (size < 0 || size > shmmax) {
  err = -EINVAL;
 } else if (key == IPC_PRIVATE) {
  err = newseg(key, shmflg, size);
 } else if ((id = findkey (key)) == -1) {
  if (!(shmflg & IPC_CREAT))
   err = -ENOENT;
  else
   err = newseg(key, shmflg, size);
 } else if ((shmflg & IPC_CREAT) && (shmflg & IPC_EXCL)) {
  err = -EEXIST;
 } else {
  shp = shm_segs[id];
  if (shp->u.shm_perm.mode & SHM_DEST)
   err = -EIDRM;
  else if (size > shp->u.shm_segsz)
   err = -EINVAL;
  else if (ipcperms (&shp->u.shm_perm, shmflg))
   err = -EACCES;
  else
   err = (int) shp->u.shm_perm.seq * SHMMNI + id;
 }
 spin_unlock(&shm_lock);
 up(&current->mm->mmap_sem);
```

```
 return err;
}
```

`shmget()` 函数的实现比较简单，首先调用 `findkey()` 函数查找值为key的共享内存是否已经被创建，`findkey()` 函数返回共享内存在 `shm_segs数组` 的索引。如果找到，那么直接返回共享内存的标识符即可。否则就调用 `newseg()` 函数创建新的共享内存。

`newseg()` 函数的实现也比较简单，就是创建一个新的 `struct shmid_kernel` 结构体，然后设置其各个字段的值，并且保存到 `shm_segs数组` 中。

shmat() 函数实现

`shmat()` 函数用于将共享内存映射到本地虚拟内存地址，由于 `shmat()` 函数的实现比较复杂，所以我们分段来分析这个函数：

```
asmlinkage long sys_shmat (int shmid, char *shmaddr, int shmflg, ulong *raddr)
{
 struct shmid_kernel *shp;
 struct vm_area_struct *shmd;
 int err = -EINVAL;
 unsigned int id;
 unsigned long addr;
 unsigned long len;

 down(&current->mm->mmap_sem);
 spin_lock(&shm_lock);
 if (shmid < 0)
  goto out;

 shp = shm_segs[id = (unsigned int) shmid % SHMMNI];
```

```
if (shp == IPC_UNUSED || shp == IPC_NOID)
 goto out;
```

上面这段代码主要通过 `shmid` 标识符来找到共享内存描述符，上面说过系统中所有的共享内存到保存在 `shm_segs` 数组中。

```
if (!(addr = (ulong) shmaddr)) {
 if (shmflg & SHM_REMAP)
  goto out;
 err = -ENOMEM;
 addr = 0;
again:
 if (!(addr = get_unmapped_area(addr, shp->u.shm_segsz))) // 获取一个空闲的虚拟内存空间
  goto out;
 if(addr & (SHMLBA - 1)) {
  addr = (addr + (SHMLBA - 1)) & ~(SHMLBA - 1);
  goto again;
 }
} else if (addr & (SHMLBA-1)) {
 if (shmflg & SHM_RND)
  addr &= ~(SHMLBA-1);       /* round down */
 else
  goto out;
}
```

上面的代码主要找到一个可用的虚拟内存地址，如果在调用 `shmat()` 函数时没有指定了虚拟内存地址，那么就通过 `get_unmapped_area()` 函数来获取一个可用的虚拟内存地址。

```
spin_unlock(&shm_lock);
err = -ENOMEM;
shmd = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
spin_lock(&shm_lock);
if (!shmd)
 goto out;
if ((shp != shm_segs[id]) || (shp->u.shm_perm.seq != (unsigned int) shmid / SHMMNI)) {
 kmem_cache_free(vm_area_cachep, shmd);
 err = -EIDRM;
 goto out;
}
```

上面的代码主要通过调用 `kmem_cache_alloc()` 函数创建一个 `vm_area_struct` 结构，在内存管理一章知道，`vm_area_struct` 结构用于管理进程的虚拟内存空间。

```
shmd->vm_private_data = shm_segs + id;
shmd->vm_start = addr;
shmd->vm_end = addr + shp->shm_npages * PAGE_SIZE;
shmd->vm_mm = current->mm;
shmd->vm_page_prot = (shmflg & SHM_RDONLY) ? PAGE_READONLY : PAGE_SHARED;
shmd->vm_flags = VM_SHM | VM_MAYSHARE | VM_SHARED
   | VM_MAYREAD | VM_MAYEXEC | VM_READ | VM_EXEC
   | ((shmflg & SHM_RDONLY) ? 0 : VM_MAYWRITE | VM_WRITE);
shmd->vm_file = NULL;
shmd->vm_offset = 0;
shmd->vm_ops = &shm_vm_ops;

shp->u.shm_nattch++;     /* prevent destruction */
spin_unlock(&shm_lock);
err = shm_map(shmd);
spin_lock(&shm_lock);
if (err)
```

```
 if (err)
  goto failed_shm_map;

 insert_attach(shp,shmd);  /* insert shmd into shp->attaches */

 shp->u.shm_lpid = current->pid;
 shp->u.shm_atime = CURRENT_TIME;

 *raddr = addr;
 err = 0;
out:
 spin_unlock(&shm_lock);
 up(&current->mm->mmap_sem);
 return err;
 ...
}
```

上面的代码主要是设置刚创建的 `vm_area_struct` 结构的各个字段，比较重要的是设置其 `vm_ops` 字段为 `shm_vm_ops`，`shm_vm_ops` 定义如下：

```
static struct vm_operations_struct shm_vm_ops = {
 shm_open,  /* open - callback for a new vm-area open */
 shm_close,  /* close - callback for when the vm-area is released */
 NULL,   /* no need to sync pages at unmap */
 NULL,   /* protect */
 NULL,   /* sync */
 NULL,   /* advise */
 shm_nopage,  /* nopage */
 NULL,   /* wppage */
 shm_swapout  /* swapout */
};
```

```
};
```

`shm_vm_ops` 的 `nopage` 回调为 `shm_nopage()` 函数，也就是说，当发生页缺失异常时将会调用此函数来恢复内存的映射。

从上面的代码可看出，`shmat()` 函数只是申请了进程的虚拟内存空间，而共享内存的物理空间并没有申请，那么在什么时候申请物理内存呢？答案就是当进程发生缺页异常的时候会调用 `shm_nopage()` 函数来恢复进程的虚拟内存地址到物理内存地址的映射。

shm_nopage() 函数实现

shm_nopage() 函数是当发生内存缺页异常时被调用的，代码如下：

```
static struct page * shm_nopage(struct vm_area_struct * shmd, unsigned long address, int no_share)
{
 pte_t pte;
 struct shmid_kernel *shp;
 unsigned int idx;
 struct page * page;

 shp = *(struct shmid_kernel **) shmd->vm_private_data;
 idx = (address - shmd->vm_start + shmd->vm_offset) >> PAGE_SHIFT;

 spin_lock(&shm_lock);
again:
 pte = shp->shm_pages[idx]; // 共享内存的页表项
 if (!pte_present(pte)) {   // 如果内存页不存在
  if (pte_none(pte)) {
   spin_unlock(&shm_lock);
   page = get_free_highpage(GFP_HIGHUSER); // 申请一个新的物理内存页
   if (!page)
```

```
    goto oom;
   clear_highpage(page);
   spin_lock(&shm_lock);
   if (pte_val(pte) != pte_val(shp->shm_pages[idx]))
    goto changed;
  } else {
   ...
  }
  shm_rss++;
  pte = pte_mkdirty(mk_pte(page, PAGE_SHARED));   // 创建页表项
  shp->shm_pages[idx] = pte;                      // 保存共享内存的页表项
 } else
  --current->maj_flt;  /* was incremented in do_no_page */

done:
 get_page(pte_page(pte));
 spin_unlock(&shm_lock);
 current->min_flt++;
 return pte_page(pte);
 ...
}
```

shm_nopage() 函数的主要功能是当发生内存缺页时，申请新的物理内存页，并映射到共享内存中。由于使用共享内存时会映射到相同的物理内存页上，从而不同进程可以共用此块内存。

--- EOF ---

推荐↓↓↓

运维

分享网络管理、网络运维、运维规划、运维开发、Python运维、Linux运维等知识，推广围绕DevOps理念的自动化运维、精益运维、...

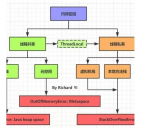
1篇原创内容

公众号

阅读原文

喜欢此内容的人还喜欢

精美图文带你掌握JVM 内存布局



顶级架构师