Lab2-Simple Share Memory

qingyunqu

Part I：

- 首先在arch/x86`/syscalls/syscall_64.tbl`中添加两个新的系统调用：320 ssmem_attach 和 321 ssmem_detach;然后在`include/linux/syscalls.h`中添加这两个函数的声明；最后在`mm/ssmem.c`中实现这两个系统调用。

初级方案：

- 初级方案先只取每个ssmem块为 1 页page frame，采用修改`mm/memory.c`中的`do_anonymous_page()`函数实现page_fault时的操作
- 向`include/linux/mm_types.h`中的`vm_area_struct`结构体中增加参数`int ssmem_id`， 0 代表不是ssmem块，不为 0 代表为ssmem块的id
- 用于存放ssmem_data的struct结构：(此处page的页数先只取一页）

```
struct ssmem_data_struct {
    size_t length;
    int count;
    struct mm_struct *list_mm[SSMEM_ATTACH_MAX];
    struct vm_area_struct *list_vma[SSMEM_ATTACH_MAX];
    struct page *page;
};
```

- 全局变量`struct ssmem_data_struct *ssmem[SSMEM_ID_MAX]`用于存放ssmem_data
- ssmem_attach开始时对传入的id做取值判断，再对length取值进行规范，再对传入的flags进行处理，下面就列出部分代码：

```
    if(id <= 0 || id >= SSMEM_ID_MAX){
        return -EINVAL;
    }
    int flags_mmap=PROT_NONE|PROT_READ;
    if(flags&SSMEM_FLAG_WRITE)
        flags_mmap=flags_mmap|PROT_WRITE;
    if(flags&SSMEM_FLAG_EXEC)
        flags_mmap=flags_mmap|PROT_EXEC;
    if(flags&SSMEM_FLAG_CRE# Lab2-Simple Share MemoryATE){
        if(ssmem[id]==NULL){
            align_length=SSMEM_PAGE_ALIGN(length);
        }
        else{
            align_length=ssmem[id]->length;
        }
    }
    else{
        if(ssmem[id]==NULL){
            return  -EADDRNOTAVALL;
        }
        else{
            align_length=ssmem[id]->length;
        }
    }
```

- 采用`do_mmap_pgoff()`函数进行ssmem块的vma分配：`addr=do_mmap_pgoff(NULL,addr,align_length,flags_mmap,MAP_PRIVATE,0,&populate);`，传入的addr参数为 0 ，代表让内核自动选取一个合适的位置进行分配，一般位于共享库的附近，MAP_PRIVATE使其为私有内存块，但也导致了copy_on_write和读时为零的问题，这些问题在之后的修改`do_anonymous_page()`函数解决

- 通过 `find_vma()` 函数找到分配的内存块
- 然后是对ssmem_data数组的初始化，其中通过kzalloc在内核地址空间为ssmem[id]分配空间,init_ssmem_data()函数为ssmem_data初始化：

```
vma->ssmem_id=id;
if(ssmem[id]==NULL){
    ssmem[id]=(struct ssmem_data_struct *)kzalloc(sizeof(struct ssmem_data_struct),GFP_KERNEL);
    ssmem[id]->length=align_length;
    ssmem[id]->count=1;
    init_ssmem_data(ssmem[id]);
    ssmem[id]->list_mm[0]=mm;
    ssmem[id]->list_vma[0]=vma;
}
else{
    int count=ssmem[id]->count;
    ssmem[id]->list_mm[count]=mm;
    ssmem[id]->list_vma[count]=vma;
    ssmem[id]->count++;
}
return addr;
```

- 修改 `mm/memory.c'中的'do_anonymous_page()'函数（在这个文件前加上` #include `来保证全局变量的导出符号）：`
  首先是解决读时为零的问题：

```
if ((!(flags & FAULT_FLAG_WRITE))&&(!(vma->ssmem_id))) {
    entry = pte_mkspecial(pfn_pte(my_zero_pfn(address),
                    vma->vm_page_prot));
    page_table = pte_offset_map_lock(mm, pmd, address, &ptl);
    if (!pte_none(*page_table))
        goto unlock;
    goto setpte;
}
```

然后解决copy_on_w rite的问题：

```
if(vma->ssmem_id){
    if(ssmem[vma->ssmem_id]->page==NULL){
        page = alloc_zeroed_user_highpage_movable(vma, address);
        ssmem[vma->ssmem_id]->page=page;
    }
    else{
        page=ssmem[vma->ssmem_id]->page;
    }
}
else{
    page = alloc_zeroed_user_highpage_movable(vma, address);
}

//page = alloc_zeroed_user_highpage_movable(vma, address);
```

以及创建反向映射页时的问题：

```
if(vma->ssmem_id){
    page_add_anon_rmap(page,vma,address);
}
else
    page_add_new_anon_rmap(page, vma, address);
```

- 经过实验可以得到如下结果：

```
root@generic_x86_64:/data/local # ./ssmpipe 1 1 writer
helloworld
```

```
root@generic_x86_64:/data/local # ./ssmpipe 1 1 reader
reader1: writer1 says helloworld
```

升级方案

- 升级方案采用挂载对应vma的vm_ops结构，实现对ssmem的vma的page_fault函数单独实现，而不修改内核的其他函数
- 挂载结构体：

```
static const struct vm_operations_struct  ssmem_ops = {
    .close = ssmem_close,
    .fault = ssmem_fault,
};
```

- 挂载的ssmem_fault函数：(该函数就简单地实现对page映射的处理）

```
static int ssmem_fault(struct vm_area_struct *vma,
            struct vm_fault *vmf)
{

    struct page *page;
    if(vma->ssmem_id){
        if(ssmem[vma->ssmem_id]->page==NULL){
            page=alloc_page(GFP_USER);
            vmf->page=page;
            ssmem[vma->ssmem_id]->page=page;

        }
        else{
            vmf->page=ssmem[vma->ssmem_id]->page;

        }
    }
    else{
        return VM_FAULT_SIGBUS;
    }
    get_page(vmf->page);
    return 0;
}
```

- 在ssmem_attach中增加`vma->vm_ops=&ssmem_ops;`来实现挂载
- 最终测试效果：

```
root@generic_x86_64:/data/local # ./ssmpipe 1 1 writer
helloworld!
```

```
root@generic_x86_64:/data/local # ./ssmpipe 1 1 reader
reader1: writer1 says helloworld!
```

ssmem_detach函数

- 功能：判断是不是ssmem内存块，如果是，再判断是不是该id的最后一个attach的进程，那么就同时释放vma和page,否则仅仅释放vma

```
if(vma==NULL){
```

```
        return -EFAULT;
    }
    else if(vma->ssmem_id==0){
        return -EFAULT;
    }
    else{
        int count;
        if(ssmem[vma->ssmem_id]->count==0)
            return -EFAULT;
        count=find_ssmem(vma->ssmem_id,mm);
        if(ssmem[vma->ssmem_id]->count==1){
            ssmem[vma->ssmem_id]->page=NULL;
        }
        do_munmap(mm,(unsigned long)ssmem[vma->ssmem_id]->list_vma[count],SSMEM_PAGE_SIZE);
        ssmem[vma->ssmem_id]->list_mm[count]=NULL;
        ssmem[vma->ssmem_id]->list_vma[count]=NULL;
        ssmem[vma->ssmem_id]->count--;
    }
```

Part II：

- 用户空间测试程序：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/syscall.h>

void ssmpipe(char*name,int ssmem_id,char*type){
    char *ssmem=NULL;
    ssmem=(char*)syscall(320,ssmem_id,7,4*1024);
    char tmp[100];
    if((strcmp(type,"writer")==0)){
        scanf("%s",tmp);
        sprintf(ssmem,"writer%s says %s",name,tmp);
    }
    else{
        printf("reader%s: %s\n",name,ssmem);
    }
    sleep(10);
    syscall(321,ssmem);
}
int main(int argc,char** argv)
{
    ssmpipe(argv[1],atoi(argv[2]),argv[3]);

    return 0;
}
```