Lab 3 Report : Simple Shared Memory

李炎 1400012951

概述：

共享内存是一种非常高效的 IPC 机制，而且实现起来相对简单，不需要复杂的进程间通信协议。本 lab 旨在通过修改安卓底层的内核代码，实现一个简单的共享内存 simple shared memory（以下简称 ssmem），不同的进程之间可以通过简单的读写共享数据。

Part 1：共享内存的抽象表示

Step 1：建立

共享内存在进程内部是通过一个 virtual memory area（以下简称 vma）来操作的，这和其他的虚拟内存区域并无二异，只不过是在不同的共享这片区域进程中都存在一个 vma 映射到同一个（或多个，决定于用户空间请求 ssmem 的长度）的物理页面。因此我们需要将所有被不同进程的 vma 共享的物理页面组织起来，将之建立成一个全局数组，便于查找和访问。同时下面会看到这个全局的数据结构在 ssmem 触发 page fault 时扮演着举足轻重的角色。
因此，定义如下结构体：

```
/* Protect ssmem array */
DECLARE_RWSEM(ssmem_mutex);

/**
 * Struct used to link tasks mapping to the same ssmem
 */
struct task_node{
    struct task_struct *task;
    struct task_node *next;
    struct task_node *prev;
    unsigned long addr; /* Virtual address in this task's virtual address space */
    struct vm_area_struct *vma; /* Vma in this task mapping to this segment */
};

/**
 * Struct used to store ssmem status
 */
struct ssmem_segment{
    struct page *page; /* Physical page mapping to this segment, initially NULL */
    int mapcnt; /* Number of task mapping to this ssmem */
    int flag; /* ssmem privilidge, create, write or exec */
    int length; /* Length of ssmem */
    int readcnt; /* Number of processes reading this shared memory */
    //DECLARE_RWSEM(rw_mutex);
    struct rw_semaphore rw_mutex; /* Mutex used to protect ssmem_segment for reader and writer */
    //DECLARE_RWSEM(cnt_mutex);
    struct rw_semaphore cnt_mutex; /* Mutex used to protect readcnt */
    struct task_node *task_head;
} *ssmem[MAX_SSMEM] = { NULL };
```

其中，ssmem_segment 是用来存储共享内存信息的，其各个字段如上图中注释所示。最重要的字段是 page，它是物理页结构体的指针，所有进程共享的物理内存即由它来表示。值得注意的是，图片顶端声明了一个读写信号量 ssmem_mutex。因为 ssmem 结构数组是全局性的，所以在访问时要保证读写同步。至于结构体中的两个读写信号量，后文还会提及。
task_node 结构体用于将映射到同一片物理内存的进程串联起来，同时其还有一个 vma 字段存储当前进程映射到这片内存的 vma。每一个 ssmem_segment 中有一个

task_node *类型的指针域 task_head，维护了一个所有共享这片内存的进程链表。
同时，我还在 vma 的结构体中添加了字段 segment_id，用以从 vma 出发查询对应的 ssmem_segment 结构体吗，类似于 linux 中的反向映射机制（reverse map）

Step 2：添加两个系统调用，创建一个共享 vma 并添加到 ssmem 数组中。

共享内存的操作是通过两个系统调用实现的。
syscall_1:
函数原型：long ssmem_attach(char *name, int flag, size_t length);
本系统调用实现过程大致包括以下几个流程：
1，参数检查和校正。对 name 的检查：name 不为 NULL；对 flag 的检查；对 length 的检查：为正数，并将其 4K 对齐。
2，创建 vma。
3，将创建的 vma 加入 ssmem 数组中。
几个关键点代码：
1，关于 id
用户空间输入的 shared memory id 默认是一个字符串，但是我们在内核中维护的是一个 ssmem 的数组，以整数作为索引。因此，有必要对用户空间输入的 name 做一个转换，即实现一个简单的 hash。原理很简单，即将字符串所有位的 asicc 码相加再取模 MAX_SSMEM 的余数。转换函数原型如下：

```
/**
 * Hash a string id to int
 * An id is an identifier to find a ssmem segment.
 * Using hash method to convert a char* id to an int id
 * can make serach convinient.
 */
static int hash_id(char *str_id){
    int len, i, asicc_code, id;

    len = strlen(str_id);
    printk("len : %d\n", len);
    asicc_code = 0;
    for(i = 0; i < len; ++i)
        asicc_code += (int)str_id[i];

    id = asicc_code % MAX_SSMEM;
    return id;
}
```

2，关于 vma 的申请
内核中提供了申请一个匿名页的函数 do_brk。但是我们不能直接使用。为了方便，我复用了 do_brk 函数的代码，添加了一个新的函数 ssmem_do_brk 函数，专门用于共享内存需要的匿名页的申请。相比普通的 do_brk，ssmem_do_brk 的改动主要有两点：
（1）对 flags 的改动：
普通的匿名页生成的 vma 节点 flags 的 VM_SHARED 位是置 0 的。也就是说，该 vma 映射到的物理内存页不允许不同进程之间共享数据。因此要在请求一个 vma 的操作中默认添加 VM_SHARED。另外，普通的 vma 申请时默认将 VM_WRITE 位置 1,也就是说 vma 默认是可以写这片内存的。我们要将之默认置 0,因为我们的共享内存使用者中既有读者又有写者，二者是区别对待的。相应代码如下：

```
/* Add VM_SHARED and not VM_WRITE defaultly */
flags = VM_DATA_DEFAULT_FLAGS | VM_ACCOUNT | VM_SHARED |mm->def_flags;
flags &= ~VM_WRITE;
```

（2）对 merge 操作的改动：
内核中申请新的虚拟地址空间片段时，先检查有没有 flag 一致的 vma 可以被合并。但我们的 ssmem 对应的 vma 是不希望被合并的。因此在新的 ssmem_do_brk 代码中将

merge 的代码注释掉：

```
/* Can we just expand an old private anonymous mapping? */
//vma = vma_merge(mm, prev, addr, addr + len, flags,
                  //NULL, NULL, pgoff, NULL, NULL);
//if (vma)
    //goto out;
```

3，ssmem 数组的维护

当新建一个 ssmem 时，对一个新申请的 ssmem 对象初始化。当 flag 中含 write 或 read 时，将当前进程通过 add_task 函数加入相应的 ssmem_segment 中。函数 add_task 部分代码如下：

```
/**
 * Add a task to ssmem segment task list (use find_task to check before add)
 */
static unsigned long add_task(int id, struct task_struct *task, int flags){
    unsigned long addr, start_brk;
    unsigned long prot;
    struct task_node *task_head, *temp;
    struct task_node *node;
    struct vm_area_struct *vma;
    struct ssmem_segment *segment;
    node = (struct task_node *)kmalloc(sizeof(struct task_node), GFP_KERNEL);
    segment = ssmem[id];

    if(segment == NULL) return 0;
    down_write(&ssmem_mutex);
    ++segment->mapcnt; /* Plus the number of task mapping to this area */
    up_write(&ssmem_mutex);
    task_head = segment->task_head;
    temp = task_head;

    start_brk = task->mm->start_brk;
    prot = PROT_READ;
    if(flags & SSMEM_FLAG_WRITE)
        prot = prot | PROT_WRITE;
    addr = ssmem_vm_brk(start_brk, segment->length);
    vma = find_vma(task->mm, addr);
    vma->segment_id = id;
    vma->vm_ops = &ssmem_mapping_vmops;
    if(flags & SSMEM_FLAG_WRITE)
        vma->vm_flags |= VM_WRITE;

    node->task = task;
    node->next = NULL;
    node->prev = NULL;
    node->addr = addr;
    node->vma = find_vma(task->mm, addr);

    while(temp->next != NULL)
        temp = temp->next;
    temp->next = node;
    node->prev = temp;
    return addr;
}
```

syscall_2:

函数原型：long ssmem_detach(void *addr);

当用户显式地调用这个调用时，内核将虚拟地址为 addr 的 vma 删除，并作一些空间回收工作。如果 vma 对应的 ssmem_segment 引用数为 0,删除该 ssmem_segment，并回收相应资源。函数原型如下：

```c
/**
 * Call by user explicitly, used to let a process detach
 * a shared memory
 */
asmlinkage long sys_ssmem_detach(void *addr){
    struct mm_struct *mm;
    struct vm_area_struct *vma;
    struct ssmem_segment *segment;
    unsigned long vaddr;
    int id;

    mm = current->mm;
    vaddr = (unsigned long)addr;
    vma = find_vma(mm, vaddr);
    if(!vma) return EEXIST;

    id = vma->segment_id;
    segment = ssmem[id];
    if(delete_vma(segment, vma))
        return EEXIST;

    return 0;
}
```

其中，detach 函数调用了 delete_vma 函数，其作用就是搜索 ssmem 数组并在必要的时候做一些删除工作，函数代码如下：

```c
/**
 * Delete a task_node node from a segment when a vma is closed or detached
 */
static int delete_vma(struct ssmem_segment *segment, struct vm_area_struct *vma){
    struct task_node *task_head, *temp, *find;
    task_head = segment->task_head;
    temp = task_head;
    find = NULL;
    while(temp != NULL){
        if(temp->vma == vma){
            find = temp;
            break;
        }
        temp = temp->next;
    }
    /* Delete find in linked list */
    if(find){
        if(!find->prev){
            if(find->next){
                find->next->prev = NULL;
                task_head = find->next;
                find->next = NULL;
            }
            else
                task_head = NULL;
        }
        else{
            if(find->next){
                find->next->prev = find->prev;
                find->prev->next = find->next;
                find->next = NULL;
                find->prev = NULL;
            }
            else{
                find->prev->next = NULL;
                find->prev = NULL;
            }
        }
        down_write(&ssmem_mutex);
        --segment->mapcnt;
        up_write(&ssmem_mutex);
        kfree(find);
    }
    else return EEXIST;
    if(segment->mapcnt == 0){
        down_write(&ssmem_mutex);
        kfree(segment);
        up_write(&ssmem_mutex);
        segment = NULL;
    }
    return 0;
}
```

Step 3：Page fault handler

在一个新的 vma 被分配之后，不会立刻分配一个物理页面，直到该虚拟页面被访问并触发

一个 page fault。因此如果要使所有在一个 ssmem_segment 中被组织起来的进程共享同一个物理页面，就要为 ssmem 注册自己的 page fault handler，使得一些 vma 访问一个已经被分配过物理页面的共享内存时，可以直接将这些 vma 映射到该物理页面。至此，我们的 ssmem_segment 结构体中的 page 页面就派上了作用。注册的新的 page fault handler 如下：

```
/**
 * Page fault handler for ssmem
 */
static int ssmem_fault(struct vm_area_struct *vma, struct vm_fault *vmf){
    int id;
    struct ssmem_segment *segment;
    struct page *page;
    printk("enter fault\n");

    id = vma->segment_id;
    segment = ssmem[id];
    page = segment->page;
    if(page){
        printk("enter if\n");
        get_page(page);
        vmf->page = page;
        return 0;
    }
    page = alloc_page(GFP_HIGHUSER_MOVABLE);
    if(!page){
        printk("can not alloc page\n");
        //return VM_FAULT_OOM;
        return VM_FAULT_SIGBUS;
    }
    else{
        printk("alloc page success\n");
        get_page(page);
        vmf->page = page;
        down_write(&ssmem_mutex);
        segment->page = page;
        up_write(&ssmem_mutex);
        return 0;
    }
    dump_stack();
    return VM_FAULT_SIGBUS;
}
```

如果在相应的 ssmem_segment 中找不到 page 字段（该字段为 NULL)，就新分配一个页面。与此同时，我也定义了相应的 close 操作：

```
/**
 * Page close handler for ssmem
 */
static void ssmem_close(struct vm_area_struct *vma){
    struct ssmem_segment *segment;
    int id;

    id = vma->segment_id;
    segment = ssmem[id];
    delete_vma(segment, vma);
}

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

二者共同构成了 ssmem 的 vma operation 集合。

在 Part 1 过程中我还编写了一些用于 debug 的函数，在找 bug 过程中起到了很好的作用：

```
static void trace_vma(struct mm_struct *mm);
static void print_segment(void);
```

Part 2：测试程序编写

Step 1：More system calls needed

第三部分要求编写用户空间函数来创建和读写共享内存。但是考虑到读写过程涉及同步机制，而用户空间实现锁无疑是繁琐的。于是干脆又开放了两个系统调用供用户使用。在这两个关于读写的系统调用中，用到了第一类读者写者问题来保证同步。即多个读者读时，写者等待，直到所有读者读完。读者写者无优先级差别。

syscall_1:

函数原型：long ssmem_write(void *addr, char *data);

函数将需要写入共享内存的数据抽象为一个字符串,通过 sprintf 操作将数据写入 经典读者写者问题 OS 课上已讲的非常清楚，此处不再赘述。函数实现如下：

```
asmlinkage long sys_ssmem_write(void *addr, char *data){
    struct task_struct *task;
    struct mm_struct *mm;
    struct vm_area_struct *vma;
    struct ssmem_segment *segment;
    unsigned long vaddr;

    vaddr = (unsigned long)addr;
    task = current;
    mm = task->mm;
    vma = find_vma(mm, vaddr);
    if(!vma)
        return EEXIST;
    segment = ssmem[vma->segment_id];

    down_write(&segment->rw_mutex);
    /* writing */
    sprintf((char *)addr, "%s", data);
    up_write(&segment->rw_mutex);

    return 0;
}
```

syscall_2:

函数原型：long ssmem_read(void *addr, char *data);

函数将需要读的数据通过 strcpy 操作拷贝到用户空间指针 data 处。函数实现如下：

```c
/**
 * Syscall used to read and write a shared memory
 * taking consideration of sync
 * Using model of first kind of reader-writer problem
 */
asmlinkage long sys_ssmem_read(void *addr, char* data){
    struct task_struct *task;
    struct mm_struct *mm;
    struct vm_area_struct *vma;
    struct ssmem_segment *segment;
    unsigned long vaddr;

    vaddr = (unsigned long)addr;
    task = current;
    mm = task->mm;
    vma = find_vma(mm, vaddr);
    if(!vma)
        return EEXIST;
    segment = ssmem[vma->segment_id];

    down_write(&segment->cnt_mutex);
    ++segment->readcnt;
    /* First reader */
    if(segment->readcnt == 1)
        down_write(&segment->rw_mutex);
    up_write(&segment->cnt_mutex);

    /* Reading */
    data = strcpy(data, (char *)addr);

    down_write(&segment->cnt_mutex);
    --segment->readcnt;
    /* Last reader */
    if(segment->readcnt == 0)
        up_write(&segment->rw_mutex);
    up_write(&segment->cnt_mutex);

    return 0;
}
```

Step 2：用户空间程序：

用户空间编写了一个程序 ssmpipe，用以测试 ssmem 的读写正确性：

```c
while(1){
    printf("input opeartion option: [create] or [write] or [read]\n");
    scanf("%s", opt);
    if(!strcmp(opt, str_create)){
        flag = 1;
        printf("input ssmem name:\n");
        scanf("%s", name);
        printf("input ssmem length(4K aligned recommanded):\n");
        scanf("%d", &length);
        addr = syscall(__NR_ssmem_attach, name, flag, length);
        if((long)addr > 0){
            printf("create success!\n");
            printf("ret : 0x%lx\n", addr);
        }
        else
            printf("create failed!\n");
    }else if(!strcmp(opt, str_write)){
        flag = 2;
        printf("input ssmem name:\n");
        scanf("%s", name);
        addr = syscall(__NR_ssmem_attach, name, flag, 1);
        if((long)addr > 0){
            printf("visit success!\n");
            printf("ret : 0x%lx\n", addr);
            printf("input your data to wirte:\n");
            scanf("%s", data);
            ret = syscall(__NR_ssmem_write, (void *)addr, data);
            printf("addr after write: %s\n", (char *)addr);
        }
        else
            printf("visit failed!\n");
    }else if(!strcmp(opt, str_read)){
        flag = 0;
        printf("input ssmem name:\n");
        scanf("%s", name);
        addr = syscall(__NR_ssmem_attach, name, flag, 1);
        if((long)addr > 0){
            printf("visit success!\n");
            printf("ret : 0x%lx\n", addr);
            ret = syscall(__NR_ssmem_read, (void *)addr, data);
            printf("read data : %s\n", data);
            printf("ret : %d\n", ret);
        }
        else
            printf("visit failed!\n");
    }
}
```

测试结果如下：

1，creator：

```
input opeartion option: [create] or [write] or [read]
create
input ssmem name:
123
input ssmem length(4K aligned recommanded):
8192
create success!
ret : 0x7effda12a000
```

2，writer：

```
input opeartion option: [create] or [write] or [read]
write
input ssmem name:
123
visit success!
ret : 0x7f1928288000
input your data to wirte:
hello!
addr after write: hello!
```

3，reader：

```
input opeartion option: [create] or [write] or [read]
read
input ssmem name:
123
visit success!
ret : 0x7f817c76f000
read data : hello!
ret : 0
```

4，kernel 中打印出的 ssmem 信息：

```
ssmem_write or ssmem_read:
segment->mapcnt : 3
segment->task->vma->vm_start : 0x7f817c76f000
segment->task->vma->vm_end : 0x7f817c771000
segment_150 :
mapcnt : 3
page : 1
task:
task_0->pid : 2330
task_0->addr : 0x7effda12a000
task_1->pid : 2335
task_1->addr : 0x7f1928288000
task_2->pid : 2340
task_2->addr : 0x7f817c76f000
end segment_150
```

结合以上调试输出，可以看到，进程号为 2330 的进程创建了一个 ssmem，相对应的 ssmem_segment 结构体存储在 ssmem 数组下标为 150 的元素处。进程号为 2335 的进程向 ssmem 中写入一个字符串“hello！”。进程号为 2340 的进程成功从共享内存中读出了该字符串。这表明我们的 ssmem 运转正常。

总结：

本 lab 中我在之前知识的基础上进一步学习了 linux 内存管理的机制，了解了虚拟内存到物理内存的映射过程，了解了 page fault handler 的执行过程，以及对 linux 的反向映射机制有了进一步的认识。如果以后有机会，我希望再将可执行文件的共享和访问加入这个 lab 中，使 ssmem 更加完善。