## Lab 3 Report : Simple Shared Memory

### 李炎 1400012951

概述：

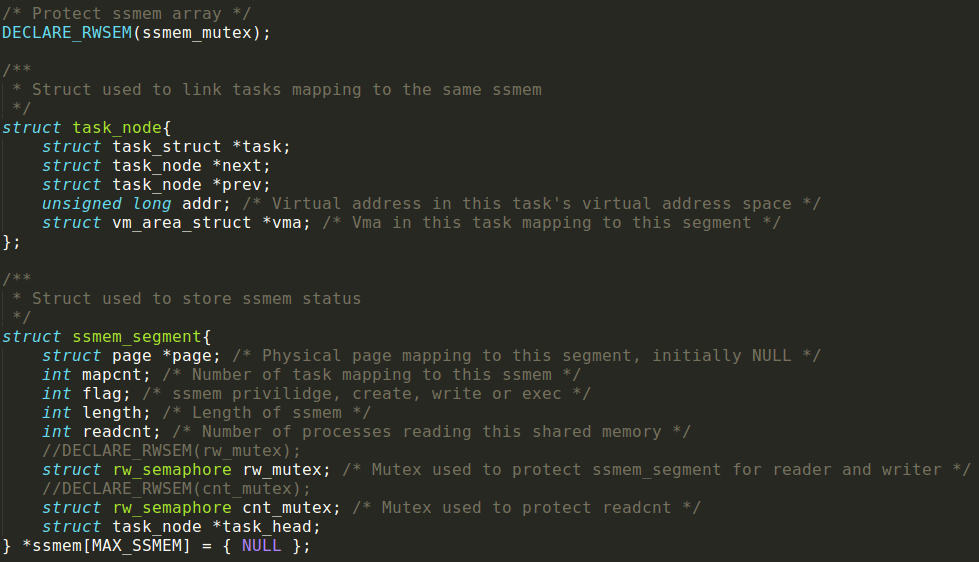
共享内存是一种非常高效的IPC机制，而且实现起来相对简单，不需要复杂的进程间通信协议。本lab旨在通过修改安卓底层的内核代码，实现一个简单的共享内存simple shared memory（以下简称ssmem），不同的进程之间可以通过简单的读写共享数据。

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：



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。

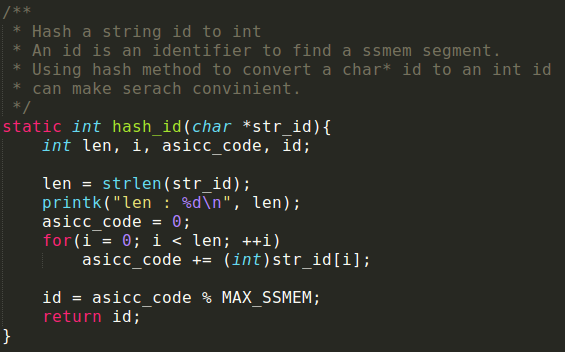
2，创建vma。

3，将创建的vma加入ssmem数组中。

几个关键点代码：

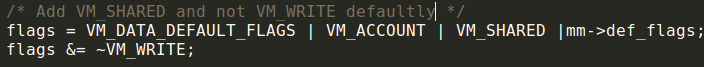
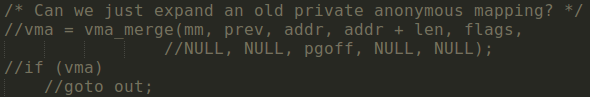
1，关于id

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：



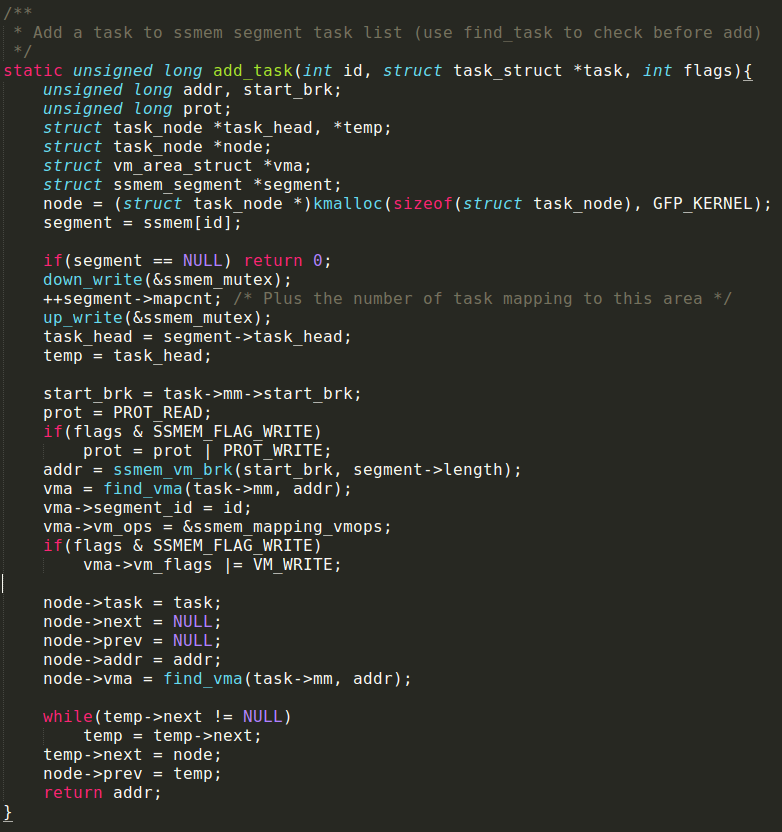
2，关于vma的申请

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：  
   
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：  
   

3，ssmem数组的维护

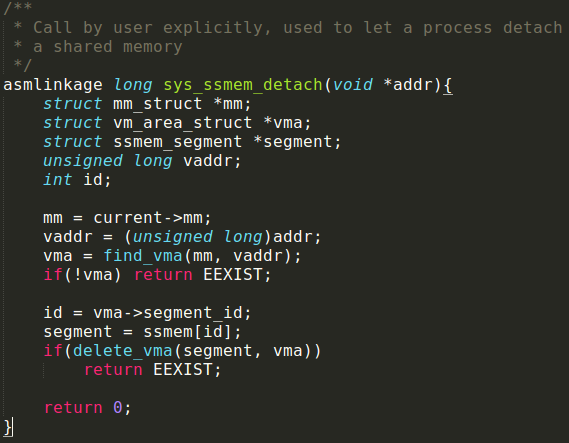
当新建一个ssmem时，对一个新申请的ssmem对象初始化。当flag中含write或read时，将当前进程通过add\_task函数加入相应的ssmem\_segment中。函数add\_task部分代码如下：



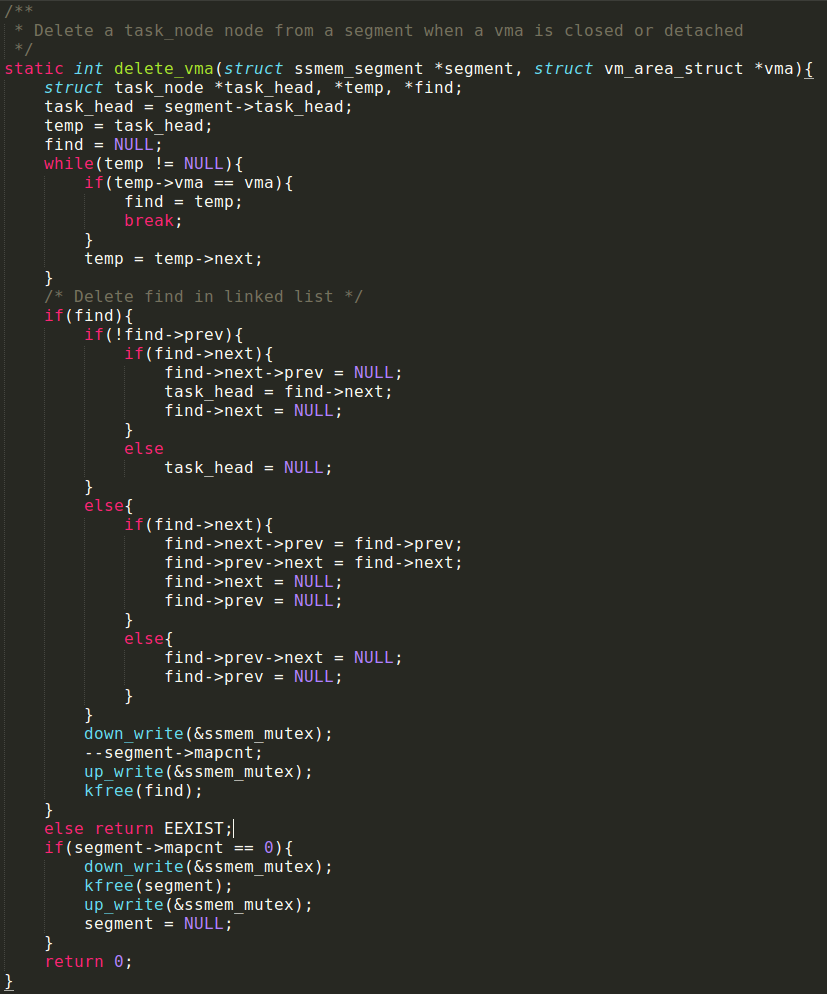
syscall\_2:

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：



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：

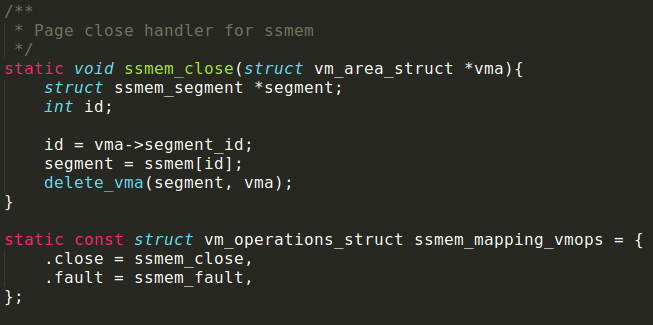


Step 3：Page fault handler

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：



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：



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：



Part 2：测试程序编写

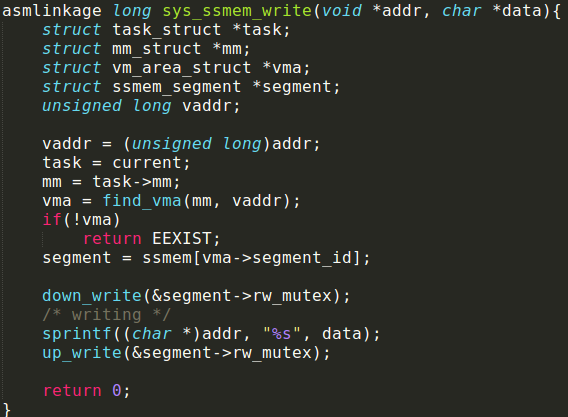
Step 1：More system calls needed

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);

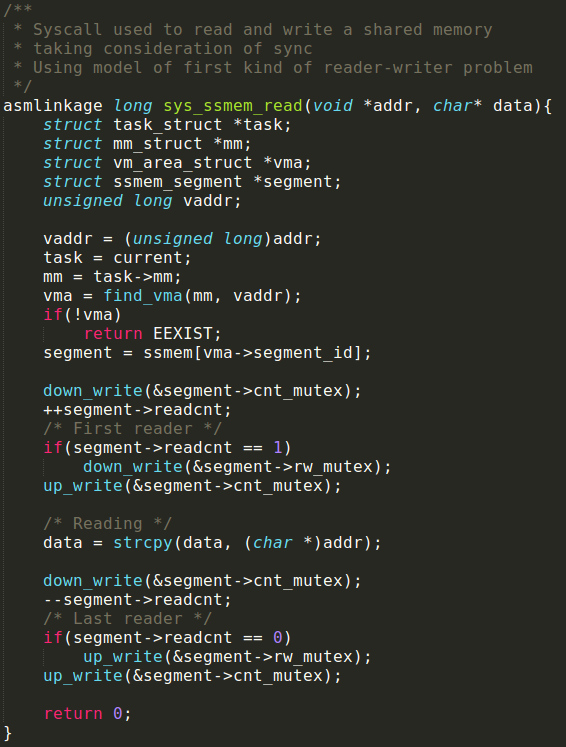
函数将需要写入共享内存的数据抽象为一个字符串,通过sprintf操作将数据写入。经典读者写者问题OS课上已讲的非常清楚，此处不再赘述。函数实现如下：



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：



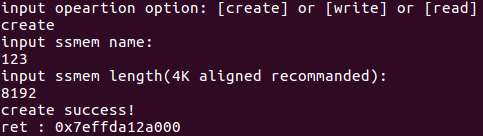
Step 2：用户空间程序：

用户空间编写了一个程序ssmpipe，用以测试ssmem的读写正确性：

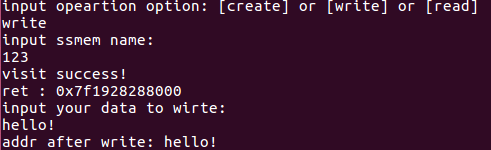


测试结果如下：

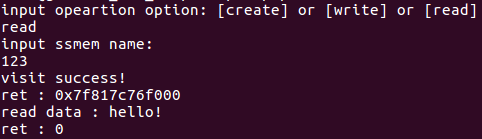
1，creator：



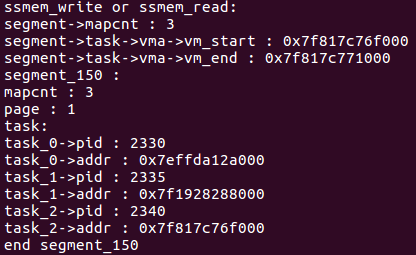
2，writer：



3，reader：



4，kernel中打印出的ssmem信息：



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。