Lab 4 Extra

准备工作：创建并切换到 `lab4-extra` 分支

请在**自动初始化分支后**，在开发机依次执行以下命令：

```
$ cd ~/学号
$ git fetch
$ git checkout lab4-extra
```

初始化的 `lab4-extra` 分支基于课下完成的 `lab4` 分支，并且在 `tests` 目录下添加了 `lab4_shm` 样例测试目录。

题目背景

在 Lab4 课下实验中，我们已经了解了系统调用机制，并且实现了一个简易的进程间通信（IPC）。但是这样的机制基于单页，并且限于双方进程，我们现在想要实现一个简易的共享内存（Shared Memory，简记为 SHM），能够实现**多进程间**的**不定页数**通信。

为了实现这一点，我们需要在**内核态**维护一个数据结构 `struct Shm`，用来维护"一块**共享内存**"，其各个成员如下：

```
struct Shm {
    u_int npage;
    struct Page *pages[N_SHM_PAGE];
    u_int open;
};
```

其中：

- `npage` 表示共享内存总共的页数，在我们的测试中被限制为 `[1, N_SHM_PAGE] = [1, 8]`
- `pages` 为用于共享的**物理页面**，通过将**不同进程的虚拟地址**映射到相同的物理页面，可以实现内存共享
- `open` 用于维护这一块共享内存的状态，`open != 0` 表示其已经被分配

在我们的测试中，限制共享内存的数量为 `N_SHM = 8` 个，使用**内核数组**维护。**数组索引即为它们各自的编号**。

系统调用约定

你需要实现下列系统调用，它们的定义与功能如下：

1. `int sys_shm_new(u_int npage)`

申请总大小为 `npage` 个页面的共享内存。在申请时，首先找到一个**编号最小的**，未被分配的（`open = 0`）的共享内存，并使用 `page_alloc` 函数申请 `npage` 个页面，进行必要的操作后，记录在 `pages` 数组中。

如果找不到这样可用的共享内存，返回 `-E_SHM_INVALID` 而无需申请页面。如果在申请页面时，空闲页面不足，返回 `-E_NO_MEM`。在这个过程中，请避免造成内存泄漏（也即，当发现分配失败时，你需要释放已经分配的页面）。

正确执行后，返回共享内存的编号（数组下标）。

2. `int sys_shm_bind(int key, u_int va, u_int perm)`

将虚拟地址 `va`（保证按页对齐）作为共享内存的起始地址，映射到编号为 `key` 的共享内存中。你需要将虚拟地址范围 `[va, va + npage * PAGE_SIZE)` **依次映射到**共享内存的 `npage` 个物理页面上。

如果对应的共享内存未被分配（`open = 0`），则返回 `-E_SHM_NOT_OPEN`。否则，返回 0 即可。

3. `int sys_shm_unbind(int key, u_int va)`

将虚拟地址范围 `[va, va + npage * PAGE_SIZE)` **解除映射**（`va` 保证按页对齐）。简单起见，这里的实现不需要关心这些虚拟地址是否真的被映射到共享内存之中，使用 `page_remove` 函数移除映射即可。

如果 `key` 对应的共享内存未被分配（`open = 0`），返回 `-E_SHM_NOT_OPEN`。否则，返回 0 即可。

4. `int sys_shm_free(int key)`

将 `key` 对应的共享内存注销（将 `open` 设为 0），并对页面进行必要的操作，将它们释放。

如果该共享内存原本未被分配，返回 `-E_SHM_NOT_OPEN`。否则，返回 0 即可。

测试数据保证在调用 `sys_shm_free` 前，全部的映射都已经被 `unbind` 了。

注意事项

1. 测试数据保证，**如果一个进程存在一个绑定了共享内存的页面时，则不会进行** `fork` ，因为此时的行为没有在题面中给出定义。你不需要考虑这种情形。
2. 你的实现要能够支持**同一个进程不同地址间**的共享，同一个进程的不同地址也可以被映射到同一个物理页面上，但测试数据保证，同一个进程的某一虚拟地址不会被多次绑定。你不需要维护虚拟地址与共享内存的关系，仅需要实现系统调用的功能即可。
3. 对于某个共享内存的页面，可能被绑定多个进程，然后被全部解除绑定，此时，你应该保证这些物理页面**仍可以进行新的绑定操作。提示：你可以通过在创建共享内存时，增加页面的** `pp_ref` **记录，并在销毁共享内存时使用** `page_decref` **函数释放页面。**
4. 测试数据保证，共享页面中的内存仅作为数组进行访问，而不会存放代码段、栈数据等特殊内容。
5. 测试程序会**使用 IPC 机制进行进程间的同步**。

题目要求

`user/lib/shm.c` 与 `include/shm.h` 的内容已在初始化分支时向仓库中添加，你可以查看它们的具体内容。

```
// include/shm.h
#ifndef _SHM_H_
#define _SHM_H_

#include <mmu.h>
#include <types.h>

#define N_SHM_PAGE 8
#define N_SHM 8

struct Shm {
    u_int npage;
    struct Page *pages[N_SHM_PAGE];
    u_int open;
};

#endif
```

```
// user/lib/shm.c
#include <lib.h>
#include <mmu.h>

int shm_new(u_int npage) {
    return syscall_shm_new(npage);
}

int shm_bind(u_int key, void *va) {
```

```
    return syscall_shm_bind(key, (u_int)va, PTE_D);
}

int shm_unbind(u_int key, void *va) {
    return syscall_shm_unbind(key, (u_int)va);
}

int shm_free(u_int key) {
    return syscall_shm_free(key);
}
```

此外，你可以按照下面的步骤进行：

1. 在 `include/error.h` 中，添加下列宏：

```
#define E_SHM_INVALID 14
#define E_SHM_NOT_OPEN 15
```

2. 在 `user/include/lib.h` 中，添加下列系统调用的用户态封装，以及库函数声明：

```
// syscalls
int syscall_shm_new(u_int npage);
int syscall_shm_bind(int key, u_int va, u_int perm);
int syscall_shm_unbind(int key, u_int va);
int syscall_shm_free(int key);

// shm.c
int shm_new(u_int npage);
int shm_bind(u_int key, void *va);
int shm_unbind(u_int key, void *va);
int shm_free(u_int key);
```

3. 在 `include/syscall.h` 中，向 `enum` 中添加以下系统调用号。请注意新增系统调用号的位置，应当位于 `MAX_SYSNO` 之前；

```
    SYS_shm_new
    SYS_shm_bind
    SYS_shm_unbind
    SYS_shm_free
```

4. 在 `user/lib/syscall_lib.c` 中添加系统调用封装的具体实现，使用 `msyscall` 函数陷入内核：

```
int syscall_shm_new(u_int npage) {
    // Lab4-Extra: Your code here. (1/8)
```

```
}

int syscall_shm_bind(int key, u_int va, u_int perm) {
    // Lab4-Extra: Your code here. (2/8)
}

int syscall_shm_unbind(int key, u_int va) {
    // Lab4-Extra: Your code here. (3/8)
}

int syscall_shm_free(int key) {
    // Lab4-Extra: Your code here. (4/8)
}
```

5. 在 `kern/syscall_all.c` 中**引入相关的头文件**，定义表示共享内存的**内核数组**，并添加系统调用在内核中的实现函数。请保证函数的定义位于系统调用函数表 `void *syscall_table[MAX_SYSNO]` 之前

```
#include <shm.h>

// ... ...

struct Shm shm_pool[N_SHM];

int sys_shm_new(u_int npage) {
    if (npage == 0 || npage > N_SHM_PAGE) {
        return -E_SHM_INVALID;
    }

    // Lab4-Extra: Your code here. (5/8)

    return ??;
}

int sys_shm_bind(int key, u_int va, u_int perm) {
    if (key < 0 || key >= N_SHM) {
        return -E_SHM_INVALID;
    }

    // Lab4-Extra: Your code here. (6/8)

    return ??;
}

int sys_shm_unbind(int key, u_int va) {
    if (key < 0 || key >= N_SHM) {
        return -E_SHM_INVALID;
    }

    // Lab4-Extra: Your code here. (7/8)
```

```
    return ??;
}

int sys_shm_free(int key) {
    if (key < 0 || key >= N_SHM) {
        return -E_SHM_INVALID;
    }

    // Lab4-Extra: Your code here. (8/8)

    return ??;
}
```

6. 在 `kern/syscall_all.c` 中的 `void *syscall_table[MAX_SYSNO]` 系统调用函数表中，为步骤 3 中添加的系统调用号添加对应的内核函数指针。

```
[SYS_shm_new] = sys_shm_new
[SYS_shm_bind] = sys_shm_bind
[SYS_shm_unbind] = sys_shm_unbind
[SYS_shm_free] = sys_shm_free
```

样例输出 & 本地测试

对于如下用户程序样例：

```
#include <lib.h>

int main() {
    volatile char *va1 = (char *)0x500000;
    volatile char *va2 = (char *)0x600000;
    int key = shm_new(2);

    shm_bind(key, (void *)va1);
    shm_bind(key, (void *)va2);

    va1[0] = va2[0] + 10;
    debugf("va1[0]: %d, va2[0]: %d, they shall be same\n", va1[0], va2[0]);

    shm_unbind(key, (void *)va1);
    shm_unbind(key, (void *)va2);

    va1[0] = va2[0] + 10;
    debugf("va1[0]: %d, va2[0]: %d, they shall be different\n", va1[0], va2[0]);

    shm_free(key);

    return 0;
```

```
}
```

应当输出：

```
va1[0]: 10, va2[0]: 10, they shall be same
va1[0]: 10, va2[0]: 0, they shall be different
```

样例说明

1. 第一行输出时， `va1` 与 `va2` 均绑定了同一块共享内存，因此对 `va1[0]` 的修改会体现在 `va2[0]` 中，它们的初始值会在分配页面时被清零。
2. 第二行输出时， `va1` 与 `va2` 首先与共享内存解绑，解绑后 `va1` 与 `va2` 不被映射到任何物理页，此时执行 `va1[0] = va2[0] + 10` ，会发生 TLB 缺失，进而触发 Lab2 中的被动页面分配机制，调用 `passive_alloc` 为 `va1` 和 `va2` 分配新的物理页面。
3. 上述测试程序只考察了单一进程下的共享内存实现，仅作为第一个测试点。

你可以使用

- `make test lab=4_shm && make run` 在本地测试上述样例（调试模式）
- `MOS_PROFILE=release make test lab=4_shm && make run` 在本地测试上述样例（开启优化）

或者在 `tests/lab4_shm/test.c` 中编写测试代码自行测试。

提交评测 & 评测标准

请在开发机中执行下列命令后，**在课程网站上提交评测**。

```
$ cd ~/学号
$ git add -A
$ git commit -m "message"  # 请将 message 改为有意义的信息
$ git push
```

在线评测时，所有的 `.mk` 文件、所有的 `Makefile` 文件、 `init/init.c` 以及 `tests/` 和 `tools/` 目录下的所有文件都可能被替换为标准版本，因此请同学们在本地开发时，**不要**在这些文件中编写实际功能所依赖的代码。

测试点和分数说明如下：

测试点序号	评测内容	分数
1	样例	5

测试点序号	评测内容	分数
2	只考察 `sys_shm_new` 和 `sys_shm_free` 两个系统调用	15
3	不考察 `sys_shm_unbind`	30
4	只考察至多两个进程间的共享内存	20
5	综合评测	30

测试点依赖关系如下（箭头方向表示依赖方向）：