For this assignment, you are given a lot of starter code,
and you only have to implement the two system calls shm_open and
shm_close. The system calls have been already added and all you have
to do is to fill in the implementation in shm.c

If you open up this file, you'll see the following data structure defined:

```
struct {
  struct spinlock lock;
  struct shm_page {
    uint id;
    char *frame;
    int refcnt;
  } shm_pages[64];
} shm_table;
```

This defines the shared memory table that we will use to keep track of
up to 64 pages of shared memory. Each page has:

-an id, this is an integer given by the program to specify the shared memory segment. Two
programs that shm_open the same id should get the same physical page.

-A pointer to the physical frame. This is a pointer to the physical page that we will share

-A reference count, which indicates the number of processes sharing this page. If we close
the shared memory region, we dont want to remove the page unless no one else is sharing it.

The assignment describes shm_open as:

"shm_open looks through the shm_table to see if this segment id already
exists. If it doesn't then it needs to allocate a page and map it, and
store this information in the shm_table. Dont forget to grab the lock
while you are working with the shm_table (why?). If the segment
already exists, increase the refence count, and use mappages to add
the mapping between the virtual address and the physical address. In
either case, return the virtual address through the second parameter
of the system call."

This is basically a full description of what you need to do. To break it down in more
detail.

1. Look through the shm_table to see if the id we are opening already exists. Two cases:

   Case 1: It already exists, which means another process did a
   shm_open before us. In this case, we find the physical address of
   the page in the table, and map it to an available page in our
   virtual address space. To map the page (i.e., add it to the page
   table) you need to use the mappages function which we saw already
   as part of allocuvm.

   mappages prototype looks like this:

```
int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
```

pgdir is the page directory pointer (which you can get from the proc
structure for your process.

va is a free virtual address you want to attach your page to (e.g.,
sz, perhaps rounded up).

size is the size you are mapping, which in our case is a single page (i.e., PGSIZE).

pa is the physical address which is the frame pointer you get from shm_table (but pass it through the V2P macro)

permissions are a set of PTE permissions. Use PTE_W|PTE_U to say the page is writeable and accessible to the user.

Finally, you increment refcnt and return the pointer to the virtual address using something like

```
*pointer=(char *)va;
```

You should also update sz since your virtual address space expanded.

Case 2: shared memory segment does not exist (not found in the table), which means we are the first process to do shm_open(). In this case, we find an empty entry in the shm_table, and initialize its id to the id passed to us. We then kmalloc a page and store its address in frame (we got our physical page). Finally, we set the refcnt to 1. At this point, the remaining implementation is similar to case 1: we map the page to an available virtual address space page (e.g., sz), and return a pointer through the pointer parameter.

That's it! Be careful to use the embedded spin lock to avoid race conditions on the shm_table (do you see how that can be a problem?). Basically, use the acquire and release calls that are used in shm_init at the appropriate places in your code.

Figuring out that you are done correctly? Run the shm_cnt program that is given to you. The last process to exit should print 20000 for the counter. Since each process increments the counter only 10000 times, this means that they successfully shared the page.

You may be tempted to use allocuvm. Unfortunately you cannot for two reasons:

1-In case 2, you dont want to allocate a physical page, only to mappages to an existing page.

2-In case 1, even though you need to allocate and map a physical page, you need a pointer to the frame. While you can dig it out from the page table, it is a bit tricky.

So, its simpler to do your own kmalloc() in case 1 where you need it, and this way you have the pointer to the frame returned from kmalloc().