Project 3b: Shared Memory in xv6

Objectives

There are two objectives to this assignment:

- To familiarize you with the xv6 virtual memory system.
- To add shared memory segments.

Overview

In this project, you'll be adding a new simple facility to allow different processes to share memory pages. Sound simple? Good! At least you think things sound simple.

Details

In most operating systems, there are some different ways for processes to communicate with one another. In this part of the project, you'll explore how to add shared-memory pages to processes that are interested in communicating through memory.

You will need to implement the following new systems calls:

void *shmgetat(int key, int num_pages): this is a simplified combination of the two Linux system calls: `shmget()` followed by `shmat()`. The idea is that if processes call **shmgetat()** with the same **key** for the first argument, then they will share the specified number of physical pages. Using different keys in different calls to **shmgetat()** corresponds to different physical pages.

On success, **shmgetat()** returns the virtual address of the shared pages to the caller, so the process can read/write them. In all cases, **shmgetat** should map the shared phyiscal pages to the next available virtual pages, starting at the high end of that process' address space. Note that different processes can have the same physical pages mapped to different locations in their virtual address space.

For example, when a process calls **shmgetat(0, 1),** the OS should map 1 physical page into the virtual address space of the caller, starting at the very high end of the address space; these pages should be zero-filled initially. The system call returns the virtual address where these pages are mapped into the caller's address space. If another process then calls **shmgetat(0, ANY_VALUE),** then this process should also get that same 1 page mapped into its virtual address space (possibly at a different virtual address). The two processes can then each read and write to this page and thus communicate. Note that the second argument to **shmgetat** is ignored if a key that has already been used is passed as the first argument; it maps the same number of pages that were specified in the first call.

Note that if a third or fourth process calls **shmgetat(0, ANY_VALUE)** then they would have that same page mapped into their address spaces as well.

However, if another key is used, then this corresponds to a new shared region. So, if any process then calls **shmgetat(1, 3)**, the OS will map 3 (new) physical pages into the address space of the calling process and associate these 3 pages with key value 1. Subsequent calls that use **key=1** will map these three pages into the calling process' address space.

In all cases, the number of pages that can be specified is small: just up to 4 pages to a single call to **shmgetat()**. Keys can range from only 0 to 7.

If any type of error ever occurs, **shmgetat** should return -1.

Another system call is needed: **int shm_refcount(int key)**. This call returns, for a particular key, how many processes currently are sharing the associated pages. Note that if a process exits, you need to decrement this reference count. If the reference count for a key goes to 0, then all state associated with those pages should be freed. Thus, if the reference count for a key goes to 0, and then a subsequent call is made to **shmgetat(key, SOME_VALUE)**, this subsequent call will be treated as a new shared segment with a new number of shared physical pages (and the pages appropriately initialized to zero).

Of course, to use shared memory carefully, one has to think about synchronization, but that is not your worry (for this project).

Some things to think about:

- Failure cases: Bad argument to system call, address space already fully in use (large heap).
- How to handle fork(): Upon fork, must also make sure child process has access to shared page, and that reference counts are updated appropriately.
- How to track reference counts to each page so as to be able to implement **shm_refcount()**.

Some clarifications (posted to class on Oct 21):

1. same process *can* call shmgetat more than once.
    1. even with the *same* key. when it is called with the *same* key, you just return the virtual address that it is already mapped at. and the reference count does not increase since it has been already mapped to this process by the very first call to shmgetat with this key.
    2. a process can definitely call shmgetat multiple times with different keys. it will then have access to multiple shared regions.
2. keys are *not* per process. they are global to the system. if process A does shmgetat(3, 1), and then process B does shmgetat(3, ANY_VALUE), they will both have access to the *same* *one* physical page.
3. when a fork is called, every shared region has to be accessible to the new process *without* it needing to call shmgetat().
4. shared regions have to be mapped starting from the *very end* of calling process's address space. example:

```
//assume no shared regions exist yet in the system

process A: shmgetat(3, 1);

//maps one physical page to the last page in calling process's
virtual address space, if the address space is not already
full.

process A: shmgetat(2, 2);

//maps two physical pages to the second to and third to last
page in calling process's virtual address space. note that the
order of shared regions in a process's virtual address space
is not correlated to value of the key.
```

The Code

The source code for xv6 (and associated README) can be found in **~cs537-1/ta/xv6/** . Everything you need to build and run and even debug the kernel is in there.

Might be good to read the xv6 book a bit: Here .

Particularly useful for this project: Chapter 1 .