COP4610: Introduction to Operating Systems
Project 5: Shared Memory for Xv6

Important Notes

- This project is optional. You will get extra credit if you do it.
- This is a **single-person** project. You may work together with other students in this class, but you must write your own code.
- Make sure your program works on `linprog.cs.fsu.edu`.
- You may be asked by the instructor or the TAs to explain your code if necessary or by random.

Overview

In this assignment, you will learn how to implement shared memory for xv6. To get started, download a **new** copy of the xv6 source code from here. Do **NOT** use the source code of the previous project.
Words of wisdom: first, please start early! This project is more challenging than the previous ones. Second, please make minimal changes to xv6; you do not want to make it hard for the TAs to grade!

Shared Memory for Xv6

Shared memory is an important inter-process communication method. It is ideal for transferring large amount of data between processes. Shared memory depends on the separation of virtual addresses from physical addresses.To implement the shared memory, the kernel maps the same physical frames into several processes, likely at different virtual addresses.

To add support of shared memory to xv6, you first need to add two system calls with syscall number `SYS_shm_open` and `SYS_shm_close` (add them to `syscall.h`). The function declarations for user-space applications to access these system calls are (add them to `user.h`):

```
int shm_open (unsigned int id);
int shm_close (unsigned int id);
```

You also need to add the necessary functions (`sys_shm_open` and `sys_shm_close`) in the kernel to handle these system calls. Please refer to project 1 and 2 for details.

Now, let me explain how to implement `sys_shm_open` and `sys_shm_close`. In this project, each shared memory page is uniquely identified by the `id` parameter (id 0 is reserved by the kernel.) For example, if two processes both call `shm_open (0xbeefbeef)`, this creates a shared page between these two processes. As such, in the kernel, you need to maintain a data structure that keeps track of the `id` and its shared physical frame. Use the following data structure for this purpose:

```
struct shm_page{
        uint    id;
        char*   frame;
```

```
        int     refcnt;
} shm_pages[64];
```

In `sys_shm_open`, you first search `shm_pages` for the `id` passed in. If the id does not already exists, you allocate a new physical frame for this `id`, and record the info in one of the empty `shm_pages` elements (i.e., an element having `id` zero). `refcnt` should be initialized to zero. Now, you have find the physical frame associated with the `id`, you can map the corresponding physical frame to the process' address space and increment the `refcnt`. `sys_shm_open` returns the virtual address that the shared page mapped at to the user space on success, and -1 on errors.

In `sys_shm_close`, you search `shm_pages` for the `id`. If found, you decrement the `refcnt`. If `refcnt` becomes zero, you release the `shm_page` by setting its id to zero. `sys_shm_close` return -1 if the `id` is invalid and 0 otherwise.

Theoretically, you need to un-map the shared page from the process when `shm_close` is called. In this project, you can skip this step. In addition, you need to update the `refcnt` when the process forks, execves, or terminates. Let's skip this step also. These simplifications cause the memory leak but makes the project much simpler.

Here are some useful hints:

- You can use `kalloc` to allocate a physical frame. `kalloc` returns the (virtual) address that the kernel can use. You need to convert it to the physical address (use `v2p`). The allocated physical frame must be cleared (use `memset`).
- As usual, the shared variable `shm_pages` needs to be protected by a lock.
- To map the shared page into a process's address space, refer to `allocuvm`. You cannot use `allocuvm` directly because it allocates a new physical frame. Instead, you need to use the shared physical frame. But the logic is essentially the same.
- Make sure your code has the proper error checking.

A User Program

In the second task of this project, you need to create a user-space program that uses the shared memory created in the first task. Like project 3, this program creates a child process by calling `fork`. After `fork`, the parent and the child both calls `shm_open` to create a shared page (Use any id you like). The parent and the child then both try to increment the second integer of the shared memory by 10,000. Because the shared memory is initialized to zero by the kernel, the correct final value of that integer should be 20,000. However, it is very unlikely that you will get 20,000 at the end of the program due to race conditions. *Access to the shared variable must be synchronized.* In this task, you also need to create a user-space spinlock using the shared memory. Define the following structure and functions in `uspinlock.h`, and implement the functions in `uspinlock.c`:

```
struct uspinlock {
        uint locked;
 }
 void uacquire (struct uspinlock *lock);
 void urelease (struct uspinlock *lock);
```

Here are some useful hints and requirements:

- Name your main program as `shm_cnt.c`.
- Because of the way shared memory is implemented, the shared memory should be created after the fork. If it is created before the fork, only one of the parent of the child may call `shm_close`. In this project, you **MUST** create the shared page after fork.

- Use the following structure in your code:

```
struct shm_cnt {
        struct uspinlock lock;
        int    cnt;
};
```

  You can convert the address returned by `shm_open` to a pointer of this structure.

- In both the parent and the child, print the value of that integer before exit. If your code is correct, at least one of these processes will print 20,000 for the shared integer.
- Refer to `spinlock.c` for how to implement `uacquire` and `urelease`. You only need to write one line of code for each function.
- Make sure your code includes the proper error checking.

Deliverables

Submit in the blackboard your modified source code of `xv6` as a `gzip` compressed tarball. Include in the tarball all files necessary for a successful build! The name of your attachment should be

`cop4610-project5-yournames.tar.gz`

with `yournames` replaced by your CS account name. Your submission will be graded by compiling and running it and reviewing the source code.

- Please make sure your source code can compile. Absolutely no credit if it does not compile.
- Please don't include the binary files. Do a `make clean` before submission. You'll make grading harder for us if you do.
- Please don't leave out any files! You'll make grading harder for us if you do.
- Please don't modify any files you don't need to! You'll make grading harder for us if you do.
- Please don't send us the meta-information from your revision control system! You'll make grading harder for us.

And if we have a hard time grading, you'll have a hard time getting points!

Useful References

☞ Xv6 book/commentary, Chapter 2 and 4
☞ Xv6 Homepage