Lab 3: Advanced Memory Management (shared memory)

Handed out August 13, 2024
Due August 30, 2024

Lab 3:

Objectives

- Implement Shared Memory

Preliminaries

For this assignment we will use some starter code which is needed for Lab 3 (the same base used for Lab 3). You can get the starter code from the lab3 github classroom link.

Implement shared memory

In this assignment you are implementing support to enable two processes to share a memory page. This is implemented by having an entry in both process page tables point to the same physical page.

Start by looking at the user program shm_cnt.c in the repo. In this program we fork a process, then both processes open a shared memory segment with the same id using:

```
shm_open(1,(char **)(&counter));
```

For this system call, the first parameter gives an id for the shared memory segment, and the pointer is used to return a pointer to the shared page. By having this pointer be of type `shm_cnt`, we can access this struct off of this pointer and we would be accessing the shared memory page.

The code then proceeds to have both processes go through a loop repeatedly incrementing the counter in the shared page (acquiring a user level spin lock to make sure we dont lose updates; test your program without the lock and see if it makes a difference). The uspinlock is implemented in the starter code in uspinlock.c and uspinlock.h -- take a look.

At the end, each process prints the value of the counter, closes the shared memory segment and exits using `shm_close(1)`. One of them should have a value of 20000 reflecting updates from both the processes. Check your code without the spinlock to see if you lose updates.

Your task is to implement shm_open and shm_close. They are already added as system calls; you should write your code in shm.c

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.

shm_close is simpler: it looks for the shared memory segment in shm_table. If it finds it it decrements the reference count. If it reaches zero, then it clears the shm_table. You do not need to free up the page since it is still mapped in the page table. Ok to leave it that way.

Bonus: mmap

Implement two system calls

`void *mmap(void *addr, size_t length)` and

`int munmap(void *addr, size_t length)`

mmap is an alternative to malloc where you also tell the operating system where to place the allocated page(s) in virtual memory space. To implement this correctly, you should first confirm that the virtual address/pages are available (otherwise error) and then allocate the physical page and add the mapping to the page directory. Specifically:

On success, `mmap()` returns a pointer to the mapped area. On error, the value `MAP_FAILED` (that is, `(void *) -1`) is returned, and errno is set to indicate the error.

unmap does the opposite and frees the memory and remove the mapping from the page table.

If you are really adventureous, you can implement the file memory mapping part of mmap (see for example mmap man page). Talk to me if you decide to do this.

Hints

IMPORTANT : Check out the Lab 3 survival guide for more detailed help.