Name: ANUJA PATIL
Student ID: 862123752

1) Implement shared memory

Approach: We want to implement support to enable two processes to share a memory page. This is implemented by having an entry in both process page tables pointing to the same physical page. We already have a test file shm_cnt.c which forks a process and each process accesses a memory page with same id. We have to write code in shm.c where we have to implement two functions – shm_open() and shm_close():

a) shm_open():

(i) We first acquire lock on the shm_table.

(ii) We then find an entry in it with the id matching with the one passed as argument.

(iii) If an entry is found, that means one process has already allocated a physical page for the memory with given id. Hence, we just have to increment the reference count of that page (to indicate how many processes are sharing this memory). We also create an entry for this in the process' page table by mapping a page. We call mappages() for current process. Size of page allocated is one page size. Virtual address for that page in the process is PGROUNDUP(currProc->sz). After mapping this page, we increment the size of stack pointer to one page size more than current one (after rounding currentProc->sz up as above). We store the virtual address of this page in a pointer passed in argument. Then we release the lock on shm_table.

(iv) If an entry in shm_table having id matching with the passed argument idis not found, then that means current process is the first one to do shm_open for memory page with given id. We are assuming that only store 64 pages can be stored because shm_pages array in shm_table has size 64. Thus, id passed as argument would be in range: 1 <= id <= 64. One way of trying to allocate physical memory page in the shm_table is to find any entry whose id is 0 in shm_pages array and allocate page for that entry. Another way is to simply allocate page at that entry which is at position (id-1) in the shm_pages array – (id-1) because arrays are 0 indexed. I preferred the second method. Hence, I set the id of entry (i-1) to argument id. Then, called kalloc() which allocates kernal memory and returns char pointer (address of allocated kernal memory) and assigned it to the frame of shm_pages[id-1]. We also set this memory to 0 using memset(). Rest is now same as in (ii) from where we call mappages(). Then we release lock on shm_table.

Following is the implementation of shm_open():

~ — apati027@sledge:~ — ssh apati027@sledge.cs.ucr.edu | apati027@sledge:~/xv6 — ssh apati027@sledge.cs.ucr.edu | ...y/UC_Riverside/Winter2020/CS1

```
int shm_open(int id, char **pointer) {
  acquire(&(shm_table.lock));
  struct proc* currProc = myproc();

  //find the id in the shm_table
  int i;
  for(i = 0; i < 64; i++) {
    if(shm_table.shm_pages[i].id == id) {
      // Another process did shm_open before current process. So, get the physical page from shm_table
      // and map it to current process's page table using mappages().
      char* mem = shm_table.shm_pages[i].frame;
      uint va = PGROUNDUP(currProc->sz);
      mappages(currProc->pgdir, (void*)va, PGSIZE, V2P(mem), PTE_W|PTE_U);
      shm_table.shm_pages[i].refcnt++;
      *pointer = (char*) va;
      currProc->sz = PGROUNDUP(currProc->sz) + PGSIZE; //update size of process' virtual address space

      release(&(shm_table.lock));
      return 0;
    }
  }

  // Did not find page with id, so current process is the first one to shm_open for this id.
  // We are assuming that 1 <= id <= 64 because there are 64 pages in physical memory.
  shm_table.shm_pages[id-1].id = id;
  shm_table.shm_pages[id-1].frame = kalloc(); //allocating physical page for new kernal memory
  memset(shm_table.shm_pages[id-1].frame, 0, PGSIZE);

  //Rest is same like the condition "if that id is found in the shm_table".
  char* mem = shm_table.shm_pages[id-1].frame;
  uint va = PGROUNDUP(currProc->sz);
  mappages(currProc->pgdir, (void*)va, PGSIZE, V2P(mem), PTE_W|PTE_U);
  shm_table.shm_pages[id-1].refcnt = 1;
  *pointer = (char*) va;
  currProc->sz = PGROUNDUP(currProc->sz) + PGSIZE; //update size of process' virtual address space

  release(&(shm_table.lock));
  return 0; //added to remove compiler warning -- you should decide what to return
```

b) shm_close():
This implementation is pretty straight-forward. We first acquire lock on shm_table. We know that physical memory page corresponding to argument id can be found at position (id-1) in the shm_pages array (because second method was preferred. In case of first method, we would have needed a loop to search for an entry with argument id in the shm_pages). We now decrement its refcnt and then check if it is greater than 0. If yes, then no worries - leave as it is. If refcnt is equal to 0, then we need to clear the shm_table. According to the lab4 problem statement, we do not need to free up the page since it is still mapped in the page table. It is okay to leave it that way. Thus, we just reset the fields of shm_pages[id-1] if its refcnt is 0 after decrementing it. Finally, we release the lock on shm_table.

Following is the implementation of shm_open():

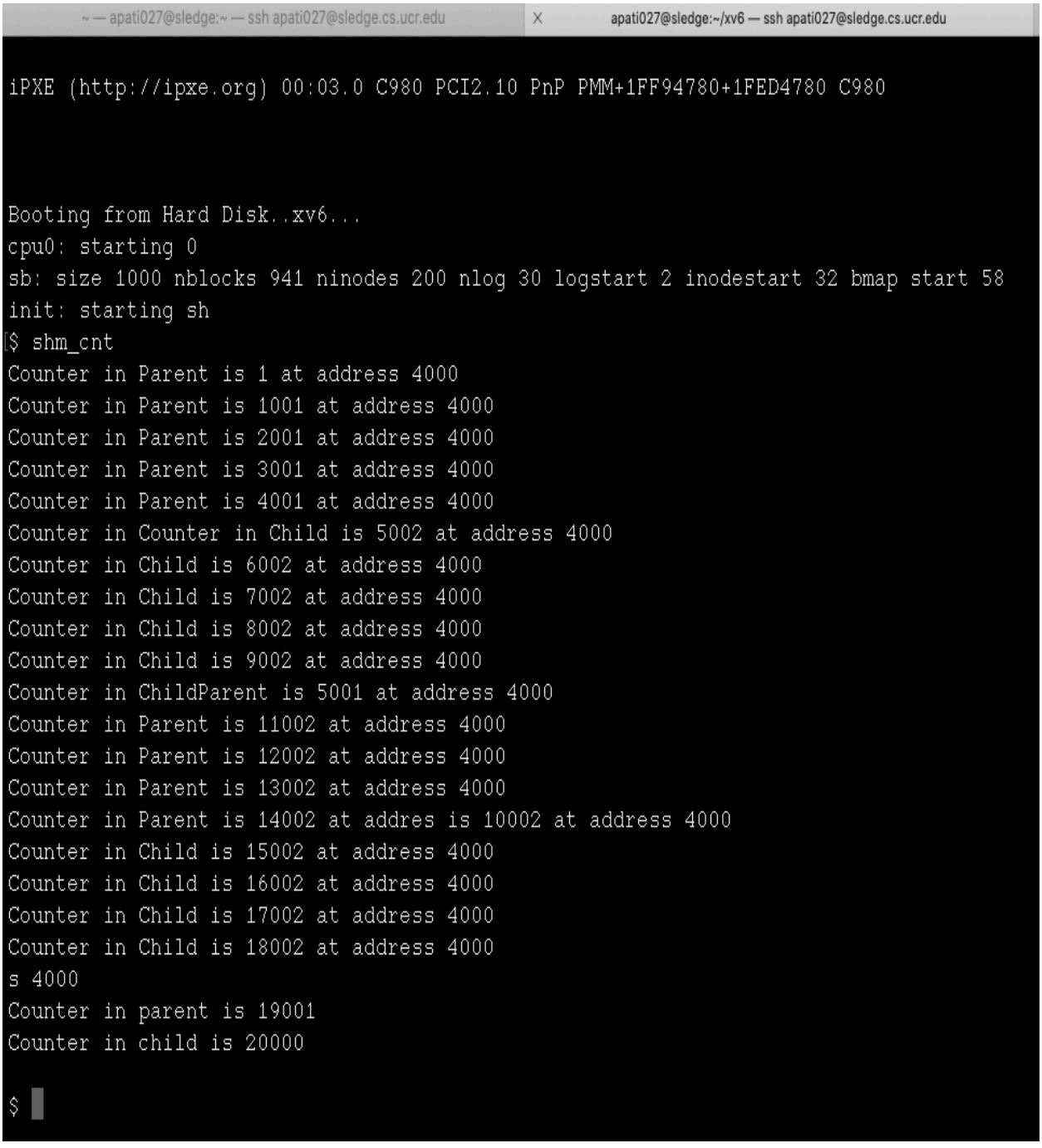
```
int shm_close(int id) {
  acquire(&(shm_table.lock));

  // We know that physical memory page corresponding to this id can be found
  // at position (id-1) in the shm_pages array. Hence, we now check its
  // refcnt. Decrement it and then check if it is greater than 0. If yes, then
  // no worries - leave as it is. If refcnt is equal to 0, then we need to clear
  // the shm_table. According to the lab4 problem statement, we do not need
  // to free up the page since it is still mapped in the page table.
  // It is okay to leave it that way.
  int i = id - 1;
  shm_table.shm_pages[i].refcnt--;
  if(shm_table.shm_pages[i].refcnt == 0) { //Reset this page
    shm_table.shm_pages[i].id = 0;
    shm_table.shm_pages[i].frame = 0;
  }

  release(&(shm_table.lock));
  return 0; //added to remove compiler warning -- you should decide what to return
}
```

2) To make and run the shm_cnt.c: (We are inside xv6 folder)
> make clean
> make qemu-nox

This will open qemu console. Then run:
$ shm_cnt
We can see how 2 processes created – child and parent are accessing same memory page (with id 1 as given in shm_cnt.c). The virtual address of the page is returned by process in shm_open()

inside the char** variable “counter”. This counter is shared between the parent and child processes. Thus, we can see that the address 4000 is same for both the processes, and the loop runs 10000 times each for parent and child. We can also see that once the counter started counting for a process, and there is a context switch to other process, then the other process will continue counting where the last process had left, indicating that it is shared. Thus, the last process to end the program has counter value 20000 and not 10000.

Following is the screenshot of output:



Summary for lab4:

The lab4 was about implementing support to enable two processes to share a memory page. We could learn how the pages are shared between processes via the shm_table and id for each memory page.