Lab 4 Report

Celyna Su (SID: 862037643) & Wei Xin Koh (SID: 862191103)

For Lab 4, we have implemented the following changes to `shm.c`:

```
31  int shm_open(int id, char **pointer) {
32    int i;
33
34    acquire(&(shm_table.lock));
35
36    // Case 1: Memory segment already exists
37    for (i = 0; i < 64; i++) {
38      if (shm_table.shm_pages[i].id == id) {
39        mappages(myproc()->pgdir, (void*) PGROUNDUP(myproc()->sz), PGSIZE, V2P(shm_table.shm_pages[i].frame), PTE_W|PTE_U);
40        shm_table.shm_pages[i].refcnt++;
41        *pointer = (char *) PGROUNDUP(myproc()->sz);
42        myproc()->sz += PGSIZE;
43        release(&(shm_table.lock));
44        return 0;
45      }
46    }
47
48    // Case 2: Shared memory segment does not exist
49    for (i = 0; i < 64; i++) {
50      if (shm_table.shm_pages[i].id == 0) {
51        shm_table.shm_pages[i].id = id;
52        shm_table.shm_pages[i].frame = kalloc();
53        shm_table.shm_pages[i].refcnt = 1;
54        memset(shm_table.shm_pages[i].frame, 0, PGSIZE);
55        mappages(myproc()->pgdir, (void*) PGROUNDUP(myproc()->sz), PGSIZE, V2P(shm_table.shm_pages[i].frame), PTE_W|PTE_U);
56        *pointer = (char *) PGROUNDUP(myproc()->sz);
57        myproc()->sz += PGSIZE;
58        release(&(shm_table.lock));
59        return 0;
60      }
61    }
62
63    release(&(shm_table.lock));
64
65    return 0;
66  }
```

In order to determine if the segment ID already exists in the `shm_table`, `shm_open` looks through the `shm_table` to search for a page that has the ID passed into the function. In order to ensure that the `shm_table` is updated by only one process at a time, we acquire the `shm_table`'s lock before performing the allocation of shared memory page to a process.

```
36    // Case 1: Memory segment already exists
37    for (i = 0; i < 64; i++) {
38      if (shm_table.shm_pages[i].id == id) {
39        mappages(myproc()->pgdir, (void*) PGROUNDUP(myproc()->sz), PGSIZE, V2P(shm_table.shm_pages[i].frame), PTE_W|PTE_U);
40        shm_table.shm_pages[i].refcnt++;
41        *pointer = (char *) PGROUNDUP(myproc()->sz);
42        myproc()->sz += PGSIZE;
43        release(&(shm_table.lock));
44        return 0;
45      }
46    }
```

[Case 1] Should the segment ID be found in the `shm_table`, the memory segment exists because another process opened said segment from a previous call to `shm_open`. Then, we get a free virtual address using `PGROUNDUP(myproc()->sz)` and find the physical address of the page in the table using `V2P(shm_table.shm_pages[i].frame)`. We then use `mappages` to map the physical address to the available page in our virtual address space. We also increment `refcnt` of the page to indicate that an additional process is referencing this page. The virtual address allocated is then stored in `*pointer` so that it can be returned through the variable `**pointer` passed into this function. Since the virtual address space expanded, we also increment the process's `sz` by `PGSIZE`. Lastly, we release the lock and return.

```
48    // Case 2: Shared memory segment does not exist
49    for (i = 0; i < 64; i++) {
50      if (shm_table.shm_pages[i].id == 0) {
51        shm_table.shm_pages[i].id = id;
52        shm_table.shm_pages[i].frame = kalloc();
53        shm_table.shm_pages[i].refcnt = 1;
54        memset(shm_table.shm_pages[i].frame, 0, PGSIZE);
55        mappages(myproc()->pgdir, (void*) PGROUNDUP(myproc()->sz), PGSIZE, V2P(shm_table.shm_pages[i].frame), PTE_W|PTE_U);
56        *pointer = (char *) PGROUNDUP(myproc()->sz);
57        myproc()->sz += PGSIZE;
58        release(&(shm_table.lock));
59        return 0;
60      }
61    }
```

[Case 2] If there is no such existing memory segment, the second for-loop will find an empty page in shm_table to allocate as shared memory segment. The page's ID will be allocated as the id passed into the function, the frame will be allocated as the kalloc's page address and the refcnt will be set as 1, to indicate that this is the first process to do shm_open. Using memset, the page the physical address is initialised to constant value 0. Next, we get a free virtual address using `PGROUNDUP(myproc()->sz)` and find the physical address of the page in the table using `V2P(shm_table.shm_pages[i].frame)`. We then use `mappages` to map the physical address to the available page in our virtual address space. The virtual address allocated is then stored in `*pointer` so that it can be returned through the variable `**pointer` passed into this function. Since the virtual address space expanded, we also increment the process's `sz` by `PGSIZE`. Lastly, we release the lock and return.

```
67  int shm_close(int id) {
68    int i;
69
70    initlock(&(shm_table.lock), "SHM lock");
71    acquire(&(shm_table.lock));
72
73    for (i = 0; i< 64; i++) {
74      if (shm_table.shm_pages[i].id == id) {
75        shm_table.shm_pages[i].refcnt--;
76
77        if (shm_table.shm_pages[i].refcnt > 0) {
78          break;
79        }
80
81        shm_table.shm_pages[i].id = 0;
82        shm_table.shm_pages[i].frame = 0;
83        shm_table.shm_pages[i].refcnt = 0;
84        break;
85      }
86    }
87
88    // Shared memory item not found
89    if (shm_table.shm_pages[i].id != id) {
90      release(&(shm_table.lock));
91      return 1;
92    }
93
94    release(&(shm_table.lock));
95
96  return 0;
97  }
```

`shm_close` looks for the shared memory segment in `shm_table`, based on the ID passed into the function. Once again, in order to ensure that the `shm_table` is updated by only one process at a time, we acquire the `shm_table`'s lock before performing the deallocation of shared memory page to a process. If it finds the shared memory segment with the specified ID, the `refcnt` of the segment is decremented. If there is still another process referencing this segment, we break out of the loop. Else, we free up the page as it is still mapped in the page table. If the ID passed into the function is not found, we release the `shm_table`'s lock and return 1, else we release the `shm_table`'s lock and return 0.

Output from running `shm_cnt`:

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ shm_cnt
Counter in Parent is 1 at address 4000
Counter in Parent is 1001 at address 4000
Counter in Parent is 2001 at address 4000
Counter in Parent is 3001 at address 4000
Counter in Parent is 4001 at address 4000
Counter in Parent is 5001 at address 4000
Counter in Parent is 6002 at address 4000
Counter in Parent is 7002 at address 4000
Counter in Parent is 8002 at address 4000
Counter in Parent is 9002 at address 4000
Counter in parent is 10001
Counter in Child is 5002 at address 4000
Counter in Child is 11001 at address 4000
Counter in Child is 12001 at address 4000
Counter in Child is 13001 at address 4000
Counter in Child is 14001 at address 4000
Counter in Child is 15001 at address 4000
Counter in Child is 16001 at address 4000
Counter in Child is 17001 at address 4000
Counter in Child is 18001 at address 4000
Counter in Child is 19001 at address 4000
Counter in child is 20000

$
```

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.