Lab 7 Semaphore Part II and XV6 System Calls

By: Sarjil Hasan
Steven Tang

Lab Objectives:

- **Understand semaphores more**
- **Experience XV6 Systems Calls**

Shared Memory:

shmget(): is used to get access to a shared memory segment by returning the ID of the shared memory segment that it creates when called.

shmat(): used to enable access to shared memory by attaching the shared memory segment identified by shmid() to the address space of the calling process.

shmdt(): detaches the shared memory segment at the address ID by smaddr().

shmctl(): changes the characteristics of a shared memory segment. It has the authority to set a new owner, group, and permissions of the memory segment and it could also destroy it too.

When we ran the program we see that we were prompted for text from shared2, which is then sent to shared1 (while it's running) through a shared memory space. So shared1 was shared2's client, as far as receiving information. Which meant that when we inputted information on the server side (shared2), we saw it on the client side (shared1).

Added common semaphore to shared1 and shared2:

```
1.  //shared1.cpp
2.  /*  After the headers the shared memory segment
3.   (the size of our shared memory structure) is created with a call to shmget,
4.   with the IPC_CREAT bit specified. It reads data from the shared memory. */
5.
6.  #include <unistd.h>
7.  #include <stdlib.h>
8.  #include <stdio.h>
9.  #include <string.h>
10. #include <semaphore.h>
11. #include <sys/stat.h>
12. #include <fcntl.h>
```

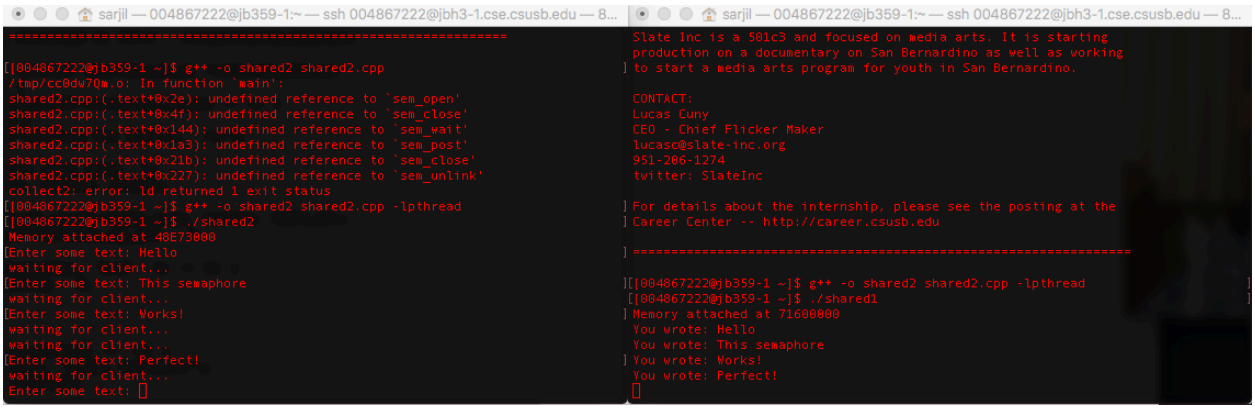
```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define TEXT_SZ 2048

struct shared_use_st {
    int written_by_you;
    char some_text[TEXT_SZ];
};

int main()
{

    char SEM_NAME[] = "foo";
    sem_t * mutex;
    mutex = sem_open(SEM_NAME, O_CREAT, 0644, 1);
    if (mutex == SEM_FAILED) {
    perror("Cannot create semaphore");
    sem_unlink(SEM_NAME);;
    exit(-1);
    }

    int running = 1;
    void *shared_memory = (void *)0;
    struct shared_use_st *shared_stuff;
    int shmid;

    srand((unsigned int)getpid());

    shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);

    if (shmid == -1) {
        fprintf(stderr, "shmget failed\n");
        exit(EXIT_FAILURE);
    }

/* We now make the shared memory accessible to the program. */

    shared_memory = shmat(shmid, (void *)0, 0);
    if (shared_memory == (void *)-1) {
        fprintf(stderr, "shmat failed\n");
        exit(EXIT_FAILURE);
    }

    printf("Memory attached at %X\n", (long)shared_memory);

/* The next portion of the program assigns the shared_memory segment to shared_stuff,
 which then prints out any text in written_by_you. The loop continues until end is found
 in written_by_you. The call to sleep forces the consumer to sit in its critical section,
 which makes the producer wait. */

    shared_stuff = (struct shared_use_st *)shared_memory;
    shared_stuff->written_by_you = 0;
    while(running) {
        if (shared_stuff->written_by_you) {
        sem_wait(mutex);
            printf("You wrote: %s", shared_stuff->some_text);
            sleep( rand() % 4 ); /* make the other process wait for us ! */
```

```
            shared_stuff->written_by_you = 0;
        sem_post(mutex);
            if (strncmp(shared_stuff->some_text, "end", 3) == 0) {
                running = 0;
            }
        }
    }
/* Lastly, the shared memory is detached and then deleted. */

    if (shmdt(shared_memory) == -1) {
        fprintf(stderr, "shmdt failed\n");
        exit(EXIT_FAILURE);
    }

    if (shmctl(shmid, IPC_RMID, 0) == -1) {
        fprintf(stderr, "shmctl(IPC_RMID) failed\n");
        exit(EXIT_FAILURE);
    }
    sem_close(mutex);
    sem_unlink(SEM_NAME);
    exit(EXIT_SUCCESS);
}
```

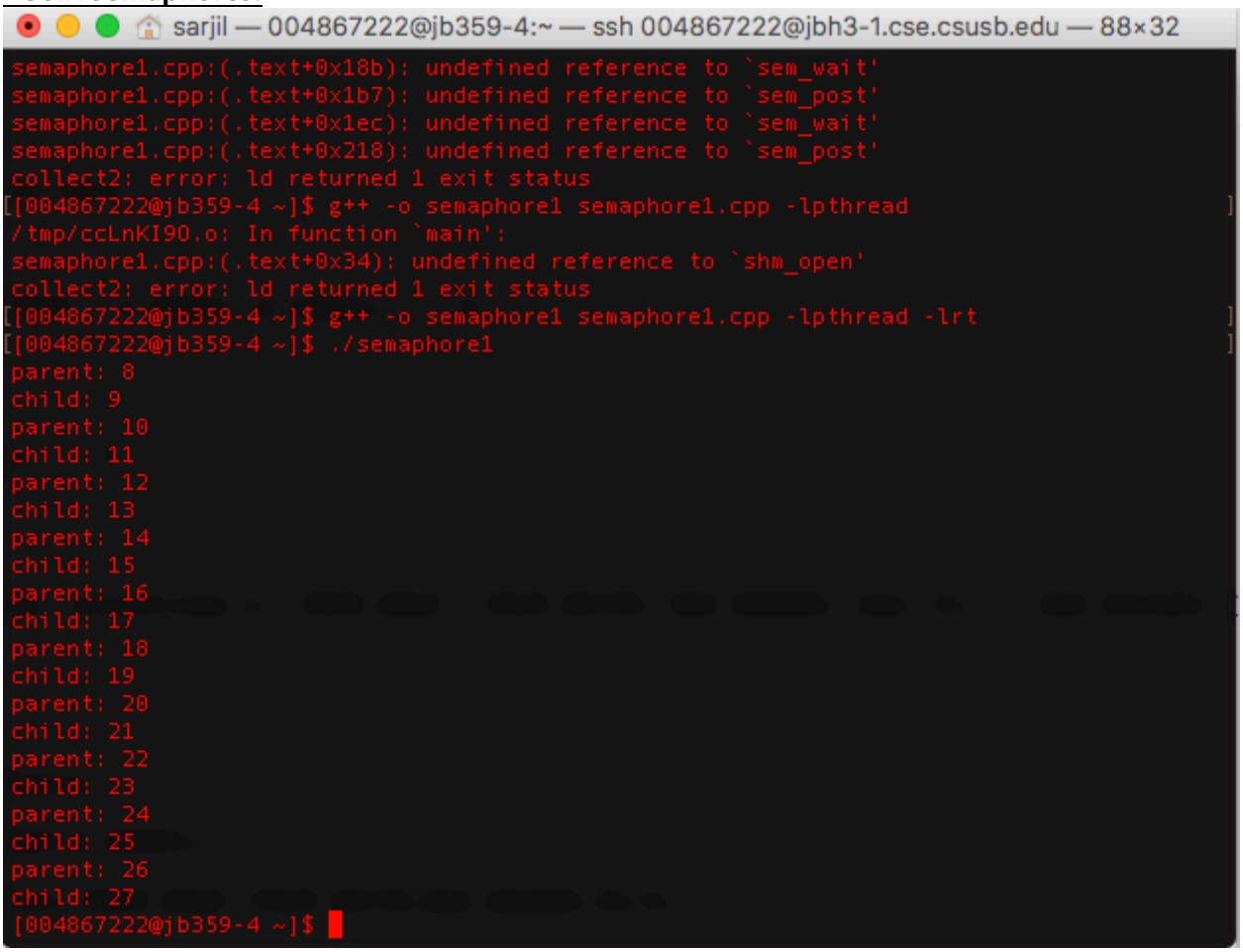
```
/*
  shared2.cpp: Similar to shared1.cpp except that it writes data to
  the shared memory.
*/
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define TEXT_SZ 2048

struct shared_use_st {
    int written_by_you;
    char some_text[TEXT_SZ];
};

int main()
{
    char SEM_NAME[] = "foo";
    sem_t * mutex;
    mutex = sem_open(SEM_NAME, 0, 0644, 0);
    if(mutex == SEM_FAILED) {
     perror("Reader: Can't access semaphore");
     sem_close(mutex);
     exit(-1);
    }

    int running = 1;
    void *shared_memory = (void *)0;
```

```
36.     struct shared_use_st *shared_stuff;
37.     char buffer[BUFSIZ];
38.     int shmid;
39.
40.     shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);
41.
42.     if (shmid == -1) {
43.         fprintf(stderr, "shmget failed\n");
44.         exit(EXIT_FAILURE);
45.     }
46.
47.     shared_memory = shmat(shmid, (void *)0, 0);
48.     if (shared_memory == (void *)-1) {
49.         fprintf(stderr, "shmat failed\n");
50.         exit(EXIT_FAILURE);
51.     }
52.
53.     printf("Memory attached at %X\n", (long)shared_memory);
54.
55.     shared_stuff = (struct shared_use_st *)shared_memory;
56.     while(running) {
57.         while(shared_stuff->written_by_you == 1) {
58.             sleep(1);
59.             printf("waiting for client...\n");
60.         }
61.     sem_wait(mutex);
62.         printf("Enter some text: ");
63.         fgets(buffer, BUFSIZ, stdin);
64.
65.         strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
66.         shared_stuff->written_by_you = 1;
67.     sem_post(mutex);
68.         if (strncmp(buffer, "end", 3) == 0) {
69.                 running = 0;
70.         }
71.     }
72.
73.     if (shmdt(shared_memory) == -1) {
74.         fprintf(stderr, "shmdt failed\n");
75.         exit(EXIT_FAILURE);
76.     }
77.     sem_close(mutex);
78.     sem_unlink(SEM_NAME);
79.     exit(EXIT_SUCCESS);
80. }
```

Typescript:

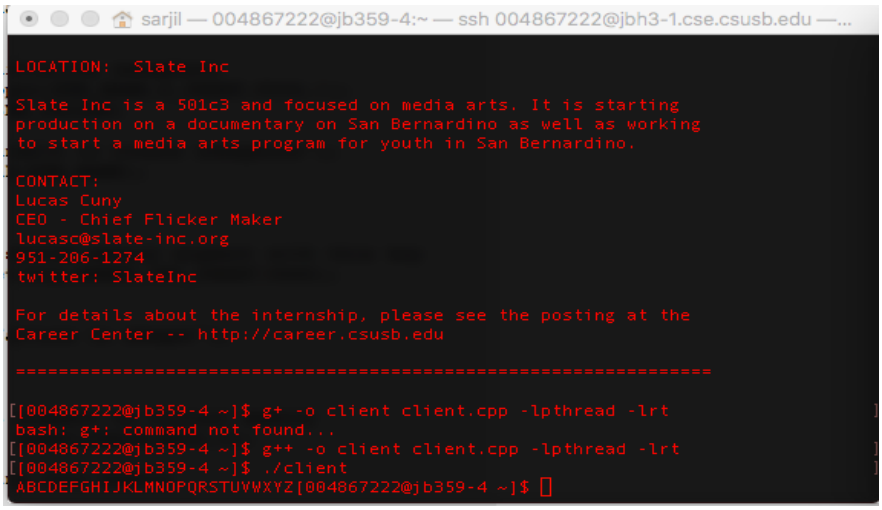


POSIX Semaphores:



What we observed when we executed this program was that each process (parent/child) went through the loop to print out a count. As we can see, the counter was incremented. The loops used shared a mutex.

Server.cpp running:



What we observed when we ran server/client was that we had to run server first because client was had to read from the array in server. If it didn't, it would read an empty array. In the array (server), it consisted of the alphabet. The client can only read the array from server because of shared memory. The reason why we run server first is because it is a semaphore. It's created in the server's file and then initialized inside of the client. So if we didn't run server first then no semaphore would have been created. There needs to be the link so we could run client after.

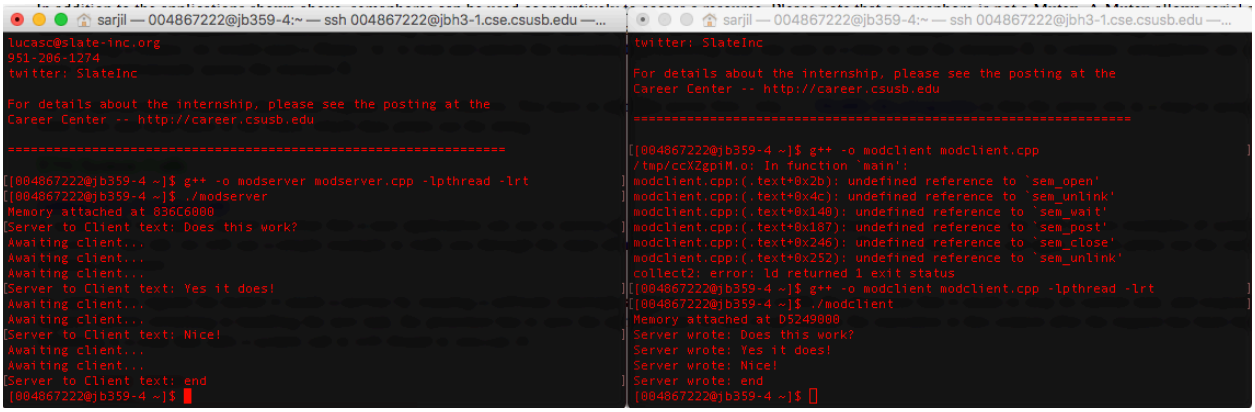
Modified server/client w/typescript:

```
1.  // modserver.cpp
2.  // g++ -o modserver modserver.cpp -lpthread -lrt
3.  #include <sys/types.h>
4.  #include <sys/ipc.h>
5.  #include <sys/shm.h>
6.  #include <stdio.h>
7.  #include <semaphore.h>
8.  #include <sys/types.h>
9.  #include <sys/stat.h>
10. #include <fcntl.h>
11. #include <unistd.h>
12. #include <stdlib.h>
13.
14. #define SHMSZ 27
15. char SEM_NAME[]= "vik";
16.
17. int main()
18. {
19.   char ch;
20.   int shmid;
21.   key_t key;
22.   char *shm,*s;
23.   sem_t *mutex;
24.
25.   //name the shared memory segment
26.   key = 1000;
27.
28.   //create & initialize semaphore
```

```
  mutex = sem_open(SEM_NAME,O_CREAT,0644,1);
  if(mutex == SEM_FAILED)
    {
      perror("unable to create semaphore");
      sem_unlink(SEM_NAME);
      exit(-1);
    }

  //create the shared memory segment with this key
  shmid = shmget(key,SHMSZ,IPC_CREAT|0666);
  if(shmid<0)
    {
      perror("failure in shmget");
      exit(-1);
    }

  //attach this segment to virtual memory
  shm = (char*) shmat(shmid,NULL,0);

 while(running) {
        while(shared_stuff->written_by_you == 1) {
            sleep(1);
            printf("Awaiting client...\n");
        }
    sem_wait(mutex);
        printf("Server to Client text: ");
        fgets(buffer, BUFSIZ, stdin);

        strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
        shared_stuff->written_by_you = 1;
    sem_post(mutex);
        if (strncmp(buffer, "end", 3) == 0) {
                running = 0;
        }
    }
  //the below loop could be replaced by binary semaphore
  while(*shm != '*')
    {
      sleep(1);
    }
  sem_close(mutex);
  sem_unlink(SEM_NAME);
  shmctl(shmid, IPC_RMID, 0);
  _exit(0);
}
```

```
// modclient.cpp
// g++ -o modclient modclient.cpp -lpthread -lrt
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

```

```c
#define SHMSZ 27
char SEM_NAME[]= "vik";

int main()
{
  char ch;
  int shmid;
  key_t key;
  char *shm,*s;
  sem_t *mutex;

  //name the shared memory segment
  key = 1000;

  //create & initialize existing semaphore
  mutex = sem_open(SEM_NAME,0,0644,0);
  if(mutex == SEM_FAILED)
    {
      perror("reader:unable to execute semaphore");
      sem_close(mutex);
      exit(-1);
    }

  //create the shared memory segment with this key
  shmid = shmget(key,SHMSZ,0666);
  if(shmid<0)
    {
      perror("reader:failure in shmget");
      exit(-1);
    }

  //attach this segment to virtual memory
    shared_stuff = (struct shared_use_st *)shared_memory;
    shared_stuff->written_by_you = 0;
    while(running) {
        if (shared_stuff->written_by_you) {
        sem_wait(mutex);
            printf("Server wrote: %s", shared_stuff->some_text);
            sleep( rand() % 4 ); /* make the other process wait for us ! */
            shared_stuff->written_by_you = 0;
        sem_post(mutex);
            if (strncmp(shared_stuff->some_text, "end", 3) == 0) {
                running = 0;
            }
        }
    }
  //once done signal exiting of reader:This can be replaced by another semaphore
  *shm = '*';
  sem_close(mutex);
  shmctl(shmid, IPC_RMID, 0);
  exit(0);
}
```



XV6 System Calls

Proc.c CPS FUNCTION:

```
1.  //current process status
2.  int
3.  cps()
4.  {
5.    struct proc *p;
6.
7.    // Enable interrupts on this processor.
8.    sti();
9.    int processCounter = 0;
10.     // Loop over process table looking for process with pid.
11.   acquire(&ptable.lock);
12.   cprintf("name \t pid \t state \n");
13.   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
14.       if ( p->state == SLEEPING ){
15.         cprintf("%s \t %d  \t SLEEPING \n ", p->name, p->pid );
16.         processCounter++;
17.       }else if ( p->state == RUNNING ){
18.         cprintf("%s \t %d  \t RUNNING \n ", p->name, p->pid );
19.         processCounter++;
20.       }
21.   }
22.
23.   release(&ptable.lock);
24.   cprintf("There are %d sleeping or running processes", processCounter);
25.   return 22;
26. }
```

PS.C

```
1.  #include "types.h"
2.  #include "stat.h"
3.  #include "user.h"
4.  #include "fcntl.h"
```

```
5.
6.  int
7.  main(int argc, char *argv[])
8.  {
9.    cps();
10.
11.   exit();
12. }
```

Typescript:

```
1.  [004532176@jb359-16 xv6-public]$ make qemu-nox
2.  dd if=/dev/zero of=xv6.img count=10000
3.  10000+0 records in
4.  10000+0 records out
5.  5120000 bytes (5.1 MB) copied, 0.118464 s, 43.2 MB/s
6.  dd if=bootblock of=xv6.img conv=notrunc
7.  1+0 records in
8.  1+0 records out
9.  512 bytes (512 B) copied, 0.00217965 s, 235 kB/s
10. dd if=kernel of=xv6.img seek=1 conv=notrunc
11. 355+1 records in
12. 355+1 records out
13. 182120 bytes (182 kB) copied, 0.00659733 s, 27.6 MB/s
14. which: no qemu in (/usr/local/bin:/opt/eclipse:/opt/scilab/bin:/opt/android-
    studio/bin:/opt/argouml:/usr/lib64/openmpi/bin:/usr/local/cuda/bin:/share/bin:/opt/Xili
    nx/14.7/ISE_DS/ISE/bin/lin64:/opt/Xilinx/14.7/ISE_DS/common/bin/lin64:/opt/android-sdk-
    linux/tools:/opt/android-sdk-linux/platform-
    tools:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/u/cse/004532176/bin/)
15. qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -
    drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
16.
17. (process:24086): GLib-
    WARNING **: gmem.c:482: custom memory allocation vtable not supported
18. xv6...
19. cpu1: starting
20. cpu0: starting
21. sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
22. init: starting sh
23. $ ps
24. name     pid     state
25. init     1       SLEEPING
26.  sh      2       SLEEPING
27.  ps      3       RUNNING
28.  There are 3 sleeping or running processes$
```

Evaluation: We have successfully completed each part of the lab. We understood each concept and while this lab was more of the challenging ones, we still completed it!

Score: 20/20