CS4375-13948 Fall 2023
Edmundo Pariente
eipariente@miners.utep.edu

Homework Report
Submitted on **12/8/2023.**

UG HW6: Semaphores for xv6

Task 3.
Implementation of sem_init(), sem_wait(), sem_post(), and sem_destroy().

sem_init(): This function first initializes a semaphore with a specific value, it will get the arguments passed by the system call such as the address, pshared and value and verify they are valid, it will then allocate space for an index of the semaphore using the function semalloc() and set the count to already provided argument.

```
int sys_sem_init(void) {
    uint64 s;
    int index, value, pshared;

        if (argaddr(0, &s) < 0 || argint(1, &pshared) < 0 || argint(2, &value) < 0)
        {
        return -1;
        }

        if (pshared == 0)
        {
        return -1;}

    index = semalloc();
    semtable.sem[index].count = value;

    if (copyout(myproc()->pagetable, s, (char*)&index, sizeof(index)) < 0)
    {
    return -1;
    }

    return 0;
}
```

sem_wait(): This function decreases the count and creates a lock on the semaphore specified by the address given, the function will execute the sleep() call when this count reaches and stays at zero, only after wake up the lock will be released.

```
int sys_sem_wait(void) {

    uint64 s;
    int addr;

  if (argaddr(0, &s) < 0 || copyin(myproc()->pagetable, (char*)&addr, s, sizeof(int)) < 0)
  {
    return -1;}

  acquire(&semtable.sem[addr].lock);

  while (semtable.sem[addr].count == 0)
  {
    sleep((void*)&semtable.sem[addr], &semtable.sem[addr].lock);
  }

    semtable.sem[addr].count--;
    release(&semtable.sem[addr].lock);

    return 0;}
```

sem_post(): This function will increment the semaphore count by first setting a lock on it, increasing it by one and then waiting for the semaphore at the given address to wake up, finally the lock will be released.

```
int sys_sem_post(void)
{
  uint64 s;
  int addr;

  if (argaddr(0, &s) < 0 || copyin(myproc()->pagetable, (char*)&addr, s, sizeof(int)) < 0)

    {
    return -1;
    }


  acquire(&semtable.sem[addr].lock);

  semtable.sem[addr].count++;
  wakeup((void*)&semtable.sem[addr]);

  release(&semtable.sem[addr].lock);

 return 0;
}
```

sem_destroy: This function will kill the semaphore by first setting the lock, then deallocate it using sedealloc() specifying the semaphore address and then release the lock.

```
int sys_sem_destroy(void) {
    uint64 s;
    int addr;

    if (argaddr(0, &s) < 0)
        {
        return -1;
       }

     acquire(&semtable.lock);

    if (copyin(myproc()->pagetable, (char*)&addr, s, sizeof(int)) < 0)
        {
            release(&semtable.lock);
            return -1;
        }

    sedealloc(addr);

    release(&semtable.lock);

    return 0;
}
```

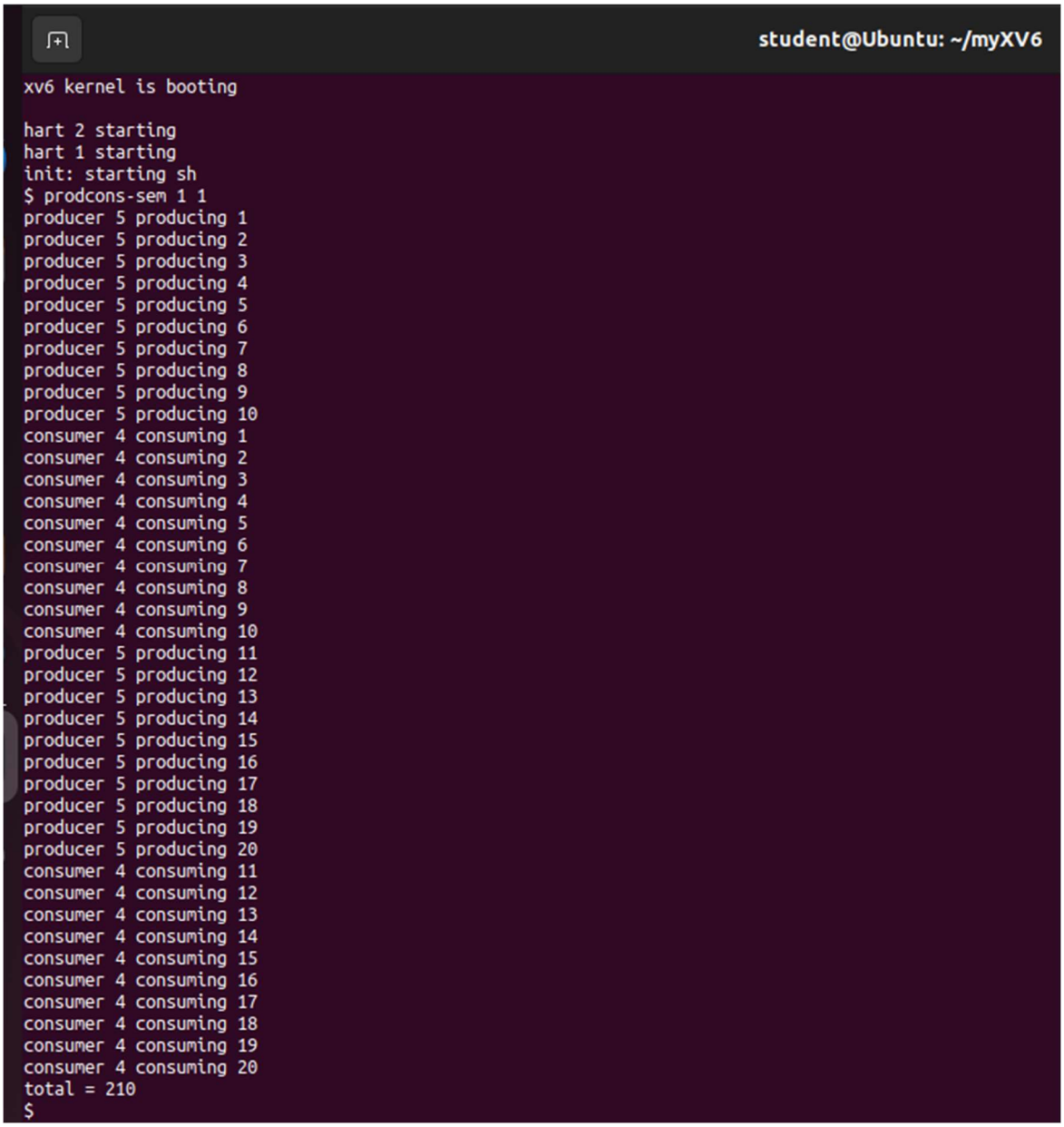
Task 4.
Test cases.

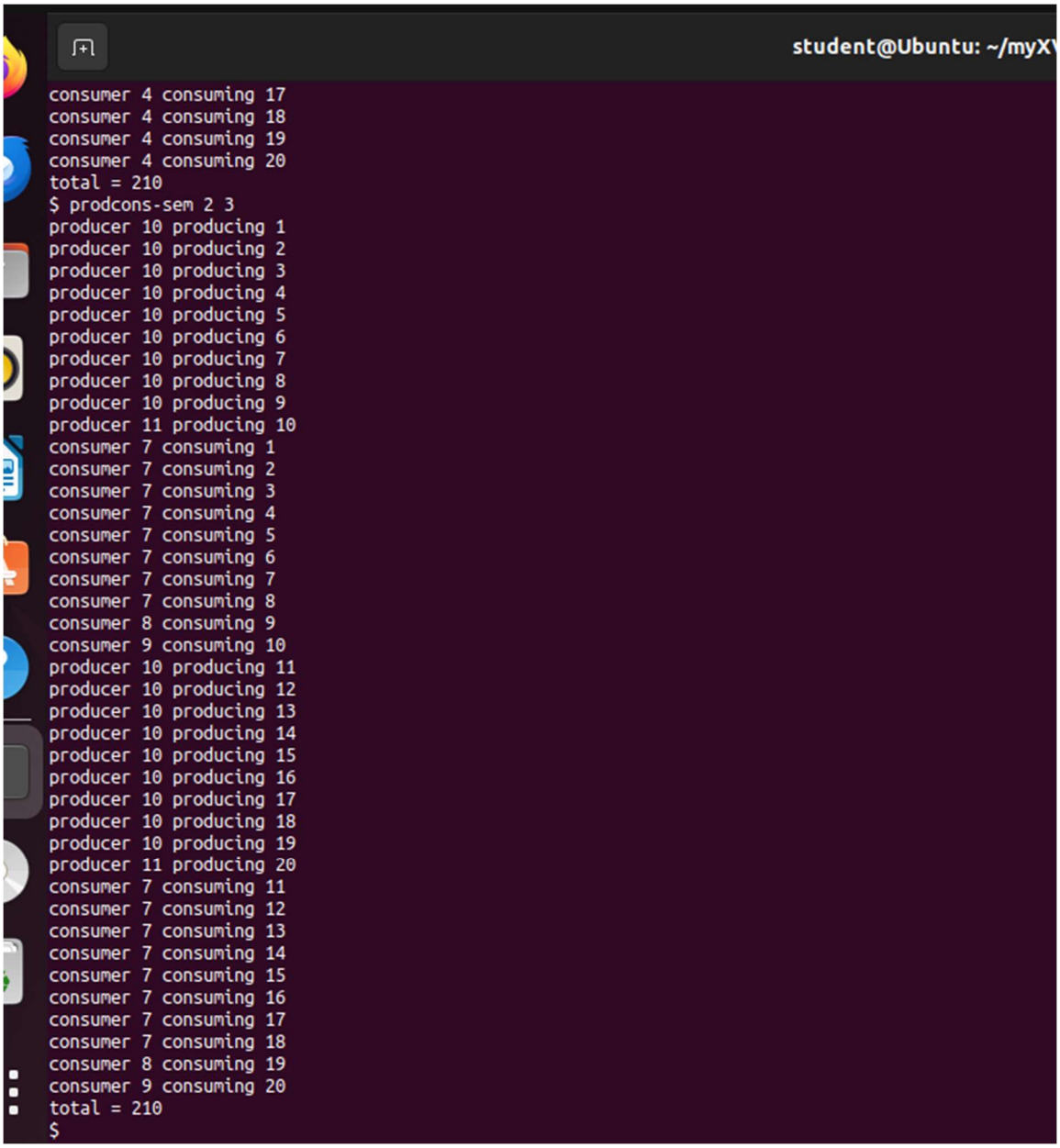
The following test cases are chosen to verify the correct behavior of the implementation of the functions in the previous task: sem_init(), sem_wait(), sem_post(), and sem_destroy().

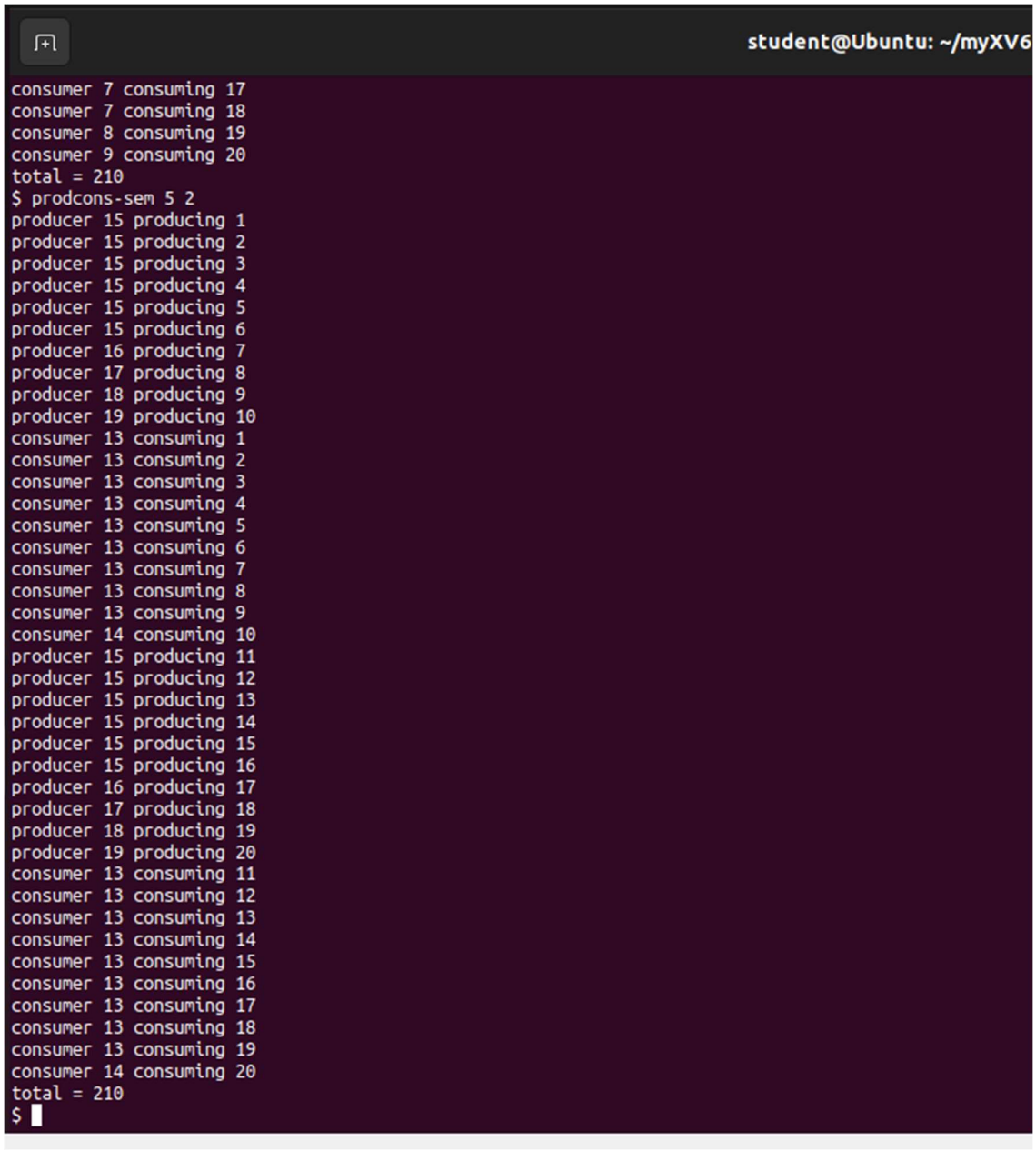
This methodology was taken from the examples that Dr. Moore provided to us to further ensure that the program was functioning as intended.

Verification of the total count variable was also conducted as the expected value was 210, so the tests had also to perform and produce such output to verify correctness of implementation.

In simple terms, each one of the newly created system calls/functions that were added to the sysproc.c file were expected to perform adequately when running these tests.


student@Ubuntu: ~/myXV6
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ prodcons-sem 1 1
producer 5 producing 1
producer 5 producing 2
producer 5 producing 3
producer 5 producing 4
producer 5 producing 5
producer 5 producing 6
producer 5 producing 7
producer 5 producing 8
producer 5 producing 9
producer 5 producing 10
consumer 4 consuming 1
consumer 4 consuming 2
consumer 4 consuming 3
consumer 4 consuming 4
consumer 4 consuming 5
consumer 4 consuming 6
consumer 4 consuming 7
consumer 4 consuming 8
consumer 4 consuming 9
consumer 4 consuming 10
producer 5 producing 11
producer 5 producing 12
producer 5 producing 13
producer 5 producing 14
producer 5 producing 15
producer 5 producing 16
producer 5 producing 17
producer 5 producing 18
producer 5 producing 19
producer 5 producing 20
consumer 4 consuming 11
consumer 4 consuming 12
consumer 4 consuming 13
consumer 4 consuming 14
consumer 4 consuming 15
consumer 4 consuming 16
consumer 4 consuming 17
consumer 4 consuming 18
consumer 4 consuming 19
consumer 4 consuming 20
total = 210
$


student@Ubuntu: ~/myX
consumer 4 consuming 17
consumer 4 consuming 18
consumer 4 consuming 19
consumer 4 consuming 20
total = 210
$ prodcons-sem 2 3
producer 10 producing 1
producer 10 producing 2
producer 10 producing 3
producer 10 producing 4
producer 10 producing 5
producer 10 producing 6
producer 10 producing 7
producer 10 producing 8
producer 10 producing 9
producer 11 producing 10
consumer 7 consuming 1
consumer 7 consuming 2
consumer 7 consuming 3
consumer 7 consuming 4
consumer 7 consuming 5
consumer 7 consuming 6
consumer 7 consuming 7
consumer 7 consuming 8
consumer 8 consuming 9
consumer 9 consuming 10
producer 10 producing 11
producer 10 producing 12
producer 10 producing 13
producer 10 producing 14
producer 10 producing 15
producer 10 producing 16
producer 10 producing 17
producer 10 producing 18
producer 10 producing 19
producer 11 producing 20
consumer 7 consuming 11
consumer 7 consuming 12
consumer 7 consuming 13
consumer 7 consuming 14
consumer 7 consuming 15
consumer 7 consuming 16
consumer 7 consuming 17
consumer 7 consuming 18
consumer 8 consuming 19
consumer 9 consuming 20
total = 210
$



Kernel bug with our implementation.

The problem with not deallocating semaphores is that it could potentially cause a system-wide resource leak and discrepancies between the value of a running process in a semaphore that was not deallocated and the actual real status of said process.

One potential solution that I can think of is to implement an automatic function that deallocates a semaphore and triggers by following the releases of the locks set in place or the returns of the internal functions.

Summary:

I definitely feel like a learned a lot of understanding into how semaphores in an operating system work and how to correctly implement them, there is a lot going on in terms of their allocation, setting a lock, sleep and wake up dynamics and how they are critical in changing behavior of the implementations. It only reinforces in my mind how complex the programming aspect in the kernel level can really get.