CS4375-13948 Fall 2023
Ashley Magallanes
avmagallanes@miners.utep.edu

Homework Report 6
Submitted on December 6, 2023

UG HW6: Semaphores for xv6

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

The implementation of the semaphore revolves around four main system calls:

1. System call for sem_init()

This method initializes the semaphore given the address passed as an argument. To do this I need to find an unused location. Therefore, here the helper methods that I implemented in the semaphore.c file for task 2 come in handy. One of the helper methods is semalloc(), which as needed it finds a semaphore index to use. This method goes through NSEM or the size defined value and checks whether each index is valid, and once it founds one that is not valid it means that is empty and it can be used, so it returns that index along with removing its valid characteristic since it will now be used. This helper method made it easier to implement the sys_sem_init().

I started by creating the data variables for each of the arguments and also one for the index being found. I then proceeded to extract the argument values for the semaphore address, the value, and the pshared form the registers. I used ardaddr() for the semaphore address and argint() for the value and pshared. After making sure all values extracted form registers were done properly then I started the initialization of the semaphore. I set the index to the return value of semalloc(). Therefore, having the variable index hold the next available semaphore index, which then I used to initialize the value. I directly accessed the count field of the semaphore by using the extern structure semtable and passed the index found above, then I set this to the value given as argument. Lastly, in order to complete the method I used copyout() to copy the semaphore form the kernel to the user space memory.

Open — sysproc.c — ~/Documents/xv6-riscv-labs-AshleyMagallanes/kernel — Save

semaphore.c | sysproc.c

```
127 int
128 sys_sem_init(void){
129         uint64 s;
130         int index;
131         int value;
132         int pshared;
133         //struct semtab semtable;
134
135         //semaphore failed
136         if(argaddr(0,&s) < 0){
137                 return -1;
138         }
139         //pshared failed
140         if(argint(1,&pshared) < 0){
141                 return -1;
142         }
143         //value failed
144         if(argint(2,&value) < 0){
145                 return -1;
146         }
147         //making sure pshared is not equal to zero
148         if(pshared == 0){
149                 return -1;
150         }
151         //initialization
152         index = semalloc();
153         semtable.sem[index].count = value;
154         //copyout
155         if(copyout(myproc()->pagetable, s, (char*)&index, sizeof(index)) <0){
156                 return -1;
157         }
158
159         return 0;
160 }
```

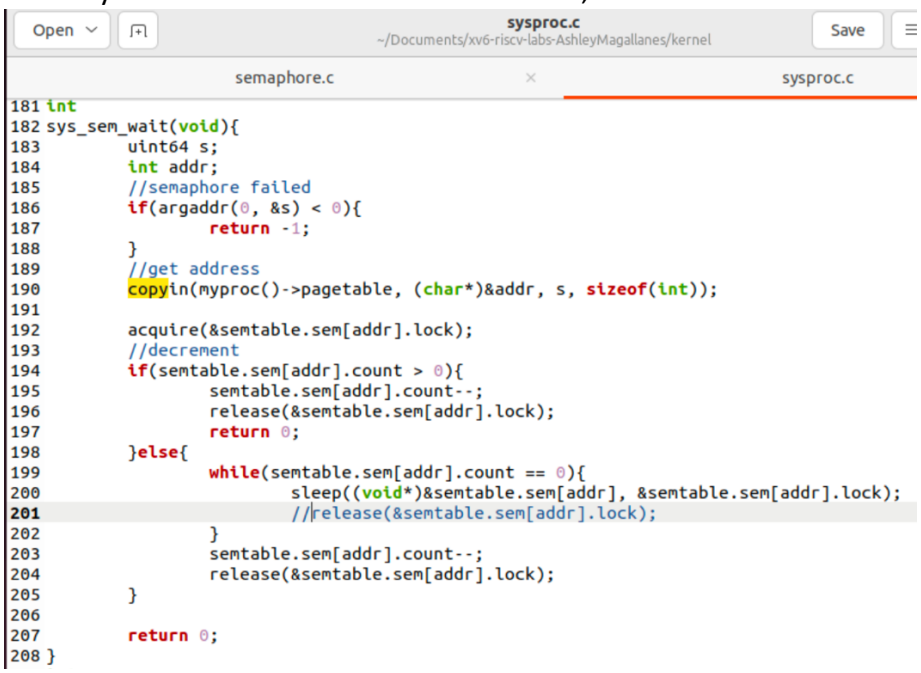
```
17 //find semaphore index to use
18 int semalloc(void){
19         acquire(&semtable.lock);
20         for (int i = 0; i < NSEM; i++){
21                 //valid means its not empty
22                 if(!semtable.sem[i].valid){
23                         //planning to use this semaphore
24                         semtable.sem[i].valid = 1;
25                         release(&semtable.lock);
26                         return i;
27                 }
28         }
29         release(&semtable.lock);
30         return -1;
31 };
32
```

2. System call for sem_wait()

This method is utilized to decrement or lock the semaphore. The already built function sleep() is useful here since the count value of the semaphore needs to be greater than zero for it to decrement. If the count value of the semaphore is equal to zero then the semaphore needs to be blocked until it is possible to decrement and that is where sleep() plays a role.

I began the implementation of sys_sem_wait() by creating the variables needed, one to hold the argument passed and another for the specific index being used of the semaphore. Then I proceeded to retrive this argument from the zero register and made sure it didn't fail. Then I used copyin() to get the address or index that would be used to access the count field of the semaphore. In addition, before any decrementing procedure I made sure to have concurrency control by adding a lock for the semaphore at the specific index being accessed.

Then as mentioned I need to decrement; therefore I checked if the count value of the semaphore at the index was greater than zero, and if so then the count was decremented and the lock was released. However, if the value of count is equal to zero then the semaphore was blocked using the sleep function. Then after the count was not zero anymore then the decrement was done, and the lock was released.



3. System call for sem_post()

This method is used to increment or unlock the semaphore. It is almost like the opposite of sem_wait(). However, for this method the count value is not required to be greater than zero in order to increment. I started the implementation just like the other two. I retrieved the semaphore address form the arguments passed by accessing the value in register zero. Then again I used copyin() to get the address or index that would be used to access the count field of the semaphore. I also made sure to have concurrency control by adding a lock for the semaphore at the specific index being

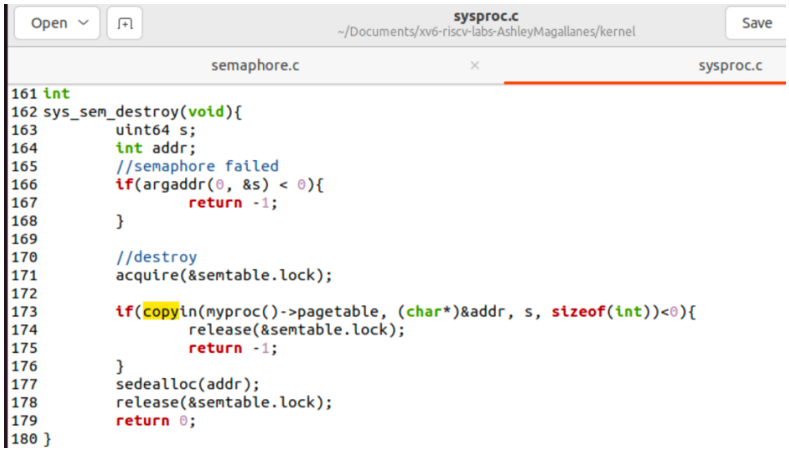
accessed. After that I incremented the semaphore count by accessing the external semtable structure and passing in the address to sem. Then I used the wakeup() function passing through the argument as the semaphore index. Lastly, I made sure to just close the lock to avoid any panic arguments.

```
209 int
210 sys_sem_post(void){
211         uint64 s;
212         int addr;
213         //semaphore failed
214         if(argaddr(0, &s) < 0){
215                 return -1;
216         }
217         //get address
218         copyin(myproc()->pagetable, (char*)&addr, s, sizeof(int));
219
220         acquire(&semtable.sem[addr].lock);
221         //increment
222         semtable.sem[addr].count++;
223         wakeup((void*)&semtable.sem[addr]);
224
225         release(&semtable.sem[addr].lock);
226
227         return 0;
228 }
```

4. System call for sem_destroy()

This method destroys the semaphore at the address passed by the argument. To do this I need to deallocate the semaphore as well. Therefore, here the helper methods that I implemented in the semaphore.c file for task 2 come in handy. One of the helper methods is sedealloc(), this function invalidates the entry of the given index from the semaphore. The method makes sure that the index passed through the argument is a valid index making sure it is in the range of size of the semtable. Then set the valid field of the semaphore index to zero, meaning that it is not used anymore. This “invalidates” the semaphore.

I implemented the sys_sem_destroy() function by getting the semaphore address of the argument from the register just like in the previous methods. Then I once again made sure to have concurrency control by adding a lock for the semaphore semtable. I used copyin() to get the index that needs to be invalidated. After getting the index I passed it through the argument of the helper method sedealloc(), which already handles the invalidation of this semaphore entry. Lastly, I just released the semtable lock.

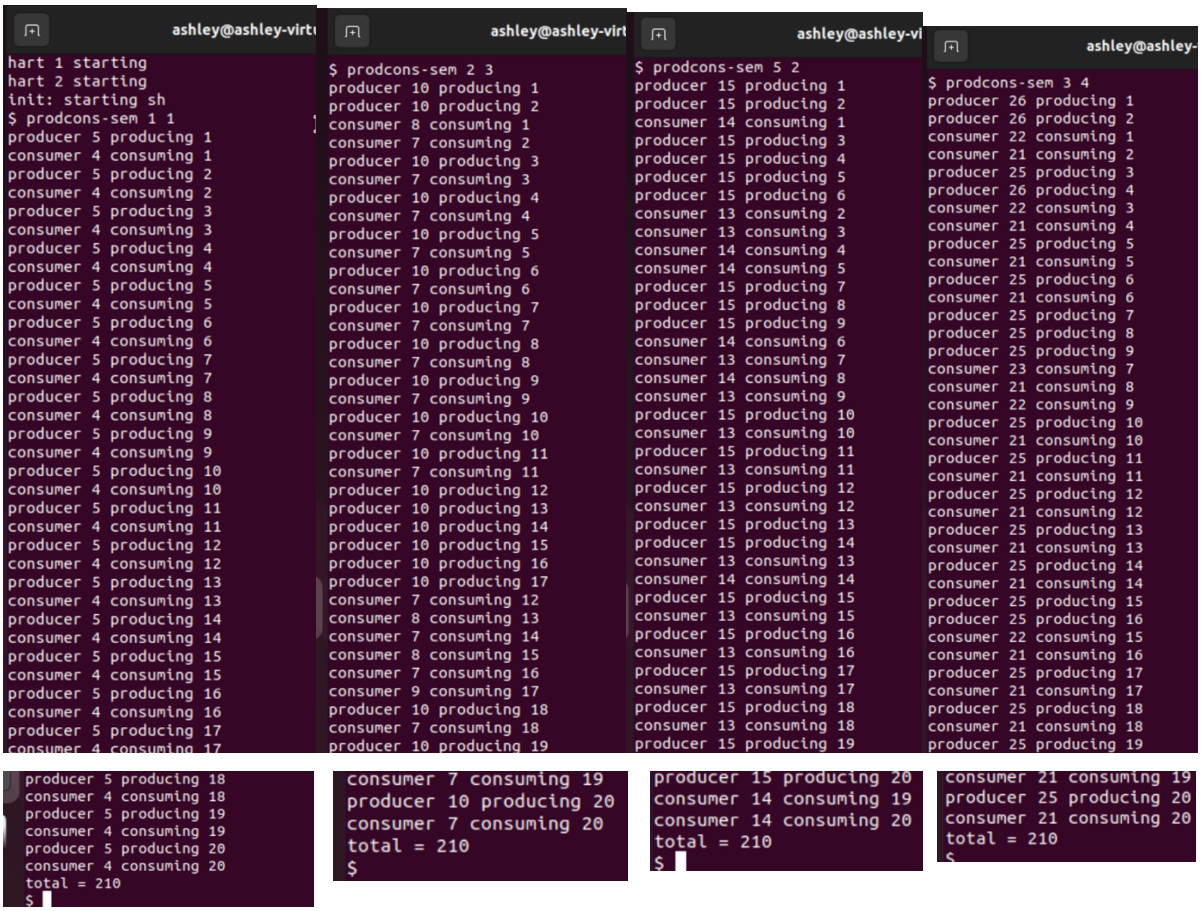


```
33 //invalidate entry of given index
34 void sedealloc(int index){
35         acquire(&semtable.sem[index].lock);
36         //make sure index arg is valid
37         if(index > -1 && index < NSEM){
38                 //invalidate the entry
39                 semtable.sem[index].valid = 0;
40         }
41         release(&semtable.sem[index].lock);
42 };
```

The hardest issue that I ran into when completing this task was panic errors in lock releases. I had to move around the locks in order to see where I needed them and where I didn't. One main issue with this was in sys_sem_wait() I was trying to release the lock both inside the while and after the whole decrement procedure, therefore I was running into race conditions. However, after some thinking I realized that only leaving the semaphore lock release after the whole procedure would work. Other than that, maybe something else was just understanding how the address accessing worked because at first I did the functions with the whole semaphore, without passing the address and of course this was giving me incorrect and inconsistent output to the testing cases of the prodcons-sem command.

Task 4.

Test cases.



The test cases above were done to demonstrate the correct implementation of the sem_init(), sem_wait(), sem_post(), and sem_destroy() methods and system calls. I decided to follow three of the output examples provided by Dr.Moore in order to make sure the output was produced properly then I also tried one more of my choice. All tests were correct and concurrent with each other with an output total of 210. Therefore, my implementation of the system calls along with the semaphore.c file was properly done.

Kernel bug with our implementation.

I followed the instructions described in the lab therefore, there is a bug in my semaphore implementation. If a user program doesn't deallocate semaphores that were allocated in the kernel semaphore table then a problem would be presented. If the operating system does not clean up after the program, meaning that it doesn't handle these deallocations of semaphores after procedures then the system can fail. If there is too many entries that are not being deallocated then this can become an issue if more processes need availability and there is not enough. This issue of the semaphore table being full of entries can also lead to failing in allocating new entries as well. All of this then an produce unexpected outputs and incorrect cases. However, this issue can be resolved by adding an implementation process to do this deallocation of unused semaphore entries.

Summary:

This homework assignment allowed me to experiment a little more with system calls but more than that I was capable of understanding threads and processes even more. I wasn't sure of why locks were needed but it makes sense that when leading with multiple threads then race conditions can happen and this leads to a shutdown of the system along with panic outputs. I believe that semaphores are important factors in the operating system, and being able to build it from scratch was really interesting!