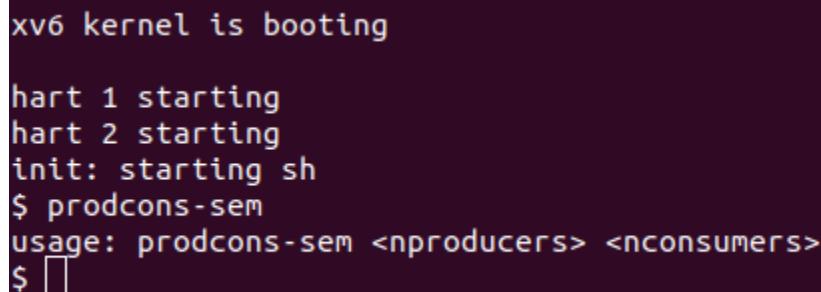


CS4375-13948 Fall 2023
Homework Report
Alberto Delgado
Submitted on 12/08/2023
adelgado@miners.utep.edu

UG HW6: Semaphores for xv6

Task 1. Semaphores are a topic in operating systems that I was not aware of prior to our discussion in class and I further gained knowledge on their functionality through this assignment. From my understanding semaphores are basically variables that are used to aid in process synchronization. Semaphores can help in limiting race conditions and establish mutual exclusion among processes. This task was just implementing the basic for this assignment. We were instructed to implement various system calls that would be used to create semaphores. There are four system calls that are needed for this assignment. I went through the process of creating these system calls. I followed the same process as previous assignments when creating system calls. Outside of what was given in the task instructions I did all necessary modifications for the system call using the `sem_t` type.

In addition to this, I also needed to re-implement the `mmap` and `mmap` functions from the previous assignment since I was having issues at the end of my assignment 5 program. This took a while to get fully functional with the new system calls but after running the proper code from the previous assignment I was able to get QEMU to execute properly. I then entered the given “`prodcons-sem.c`” file and entered it as a user command and got the following output.



```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ prodcons-sem
usage: prodcons-sem <nproducers> <nconsumers>
$
```

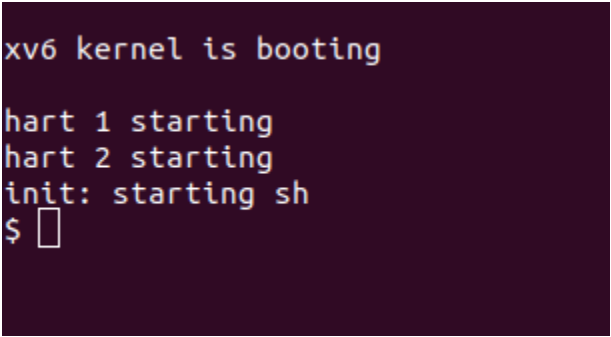
As we can see the command worked but nothing was displayed since the system call has not been implemented. This is what we are instructed to do in the next few tasks.

Task 2. For the second task that we were assigned, we were asked to implement the semaphore structures to keep track of the semaphores. I

simply followed the instructions that were given to us for the first portion which included updating files with new lines of code. We were also given the task to implement semaphore functions that would be used for the allocating and deallocating semaphores. We were given the `seminit()` function which is responsible for initializing the table and locks associated with the semaphore that are being entered into the table. So for each process that is trying to create semaphore, this is where the process would begin for the semaphore creation.

The next function was the `semalloc()` function which is responsible for allocating a semaphore from the table. The way that I went about creating the code was I first established a lock which locks the table to ensure exclusivity. With the for loop, the table would be iterated through to find the first available semaphore entry. The if statement is used to check the lock of the current available semaphore entry that was detected during the iteration. If the available semaphore is not locked then the semaphores entry index is returned. Once this entire process is complete then the lock is released.

The final function was the `semdalloc()` function which does the reversal of the previous function. This function helps in releasing the the semaphore from a specific index. I still used locks which would ensure that the process is executed safely. The if statement will check that the bounds of the desired index is within the bounds of the table, if not then the process will simply be released. If a valid index is entered then the semaphore at that specific index will be released. Once this entire process is complete then the lock is released. Once I complete all this I did all appropriate modifications and ran "make qemu", below is the output that I received.

A screenshot of a terminal window with a dark purple background. The text is displayed in a light blue/cyan monospaced font. The output shows the xv6 kernel booting, followed by two harts starting, the init process starting a shell, and a prompt character '\$' followed by a small square cursor.

```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ □
```

All compilation errors and issue were resolved and I was able to successfully compile and run qemu.

Task 3. This task focused on implementing functionality into the system call declarations of task one. The point of this is to give user functionality to the system calls that were initialized. The first functions that I worked on was the `sys_sem_init()` function which will initialize the semaphore that was entered by the user. The first if statement I used to obtain the value from the user to enter into the index variable that I initialized. Once the index is retrieved

then the index is checked to see if it is in bounds, if not then the process is ended. Once the index is validated then a lock is acquired, once a lock is obtained then I call the `sem_init()` function from the previous task to execute. Once the process is complete then the lock is released.

The next function was the `sys_sem_wait()` function which is used to wait on a semaphore. The first if statement will get the index from the user, and the following if statement will check that the index is within bounds. Once it is validated then I initialize a lock for the process. I then use a while loop which will be used to continually see if there is a need to continue waiting on the semaphore. If the process is killed then a "-1" is returned to exit the while loop. The second if statement within the while loop will check if the count is greater than zero, which if it is then the count will be decremented and releases the lock which will allow the process to continue. If the count is 0 then the process is put to sleep. The reason I use sleep here is because when the semaphore is not available then the process would continue to use resources. To avoid this we first release the lock to avoid any starvation or dead locks. Once the lock is released then the semaphore is put to sleep to free up resources and will be woken up only with the `wakeup()` function which is implemented in the next function.

The next function is the `sys_sem_post()` function which as stated previously contains the `wakeup()` function. This function is similar to the other in that it contains the same index checks. Aside from this, the function focuses on waking up any sleeping semaphores. I implement a lock and a count. The reason that I use the count in this function is to use as an argument for the `wakeup()` function. The count variable that is initialized gets the address of the associated semaphore. This function is important since semaphore may need to be woken up to continue being process without withing and absorbing resources.

The final function was the `sys_sem_destroy()` function which is used to do the opposite of `sys_sem_init()`. This function focuses on deallocating the semaphore that is specified by the user. Similarly to the other functions, this function gets the index from the user and verifies that the entered index is valid. Once this is completed the function works exactly as the `sys_sem_init()` function with the only change being that instead of calling the `sem_init()` function I call the `semdealloc()` to deallocate the semaphore specified by the user.

Task 4. Unfortunately I was unable to get my program from assignment 5 fully functional so I was unable to test the programs and even then testing the producer and consumer way as stated in the instructions I continually ran into issues as well as shown below.

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ prodcons-sem 1 1
panic: mappages: size
█
```

This is a very similar output as to the issue that I got in the previous assignment. Although I was having issues I will be explaining the example output to the best of my ability as to not let the errors in my code hinder me. The example that I will discuss is the second example in the handout. The way that producers and consumers work is simple, producers produce items and consumers consume the items produced. In the example with "2 3" the first line is a producer of 10 producing 1, the following is a consumer 7 consuming 1. This is a pattern that continues through the example and in all the examples the total is 210 and there is no producer or consumer exceeding 20. The reason is that in the "prodcons-sem.c" file, the MAX is 20 so there is no exceeding 20. So as the producer produces, the consumer consumes as evident in the examples.

Kernel bug with our implementation.

Although I was unable to test for any bugs or verify if I was able to remove the bug, I will continue this section by explaining the problems that can occur if semaphores are not deallocated. The biggest one is the occupation of valuable resources. When a semaphore is allocated that takes a certain amount of resources that would be used elsewhere, if the semaphore is not deallocated then the resources will be used and new processed may never obtain the desired semaphore. On top of this just how memory leaks can occur when not properly deallocated, semaphores that are not deallocated can cause resource leaks. Just as it sounds when there are no resources to share among programs, there can also be a drop in overall performance since the hardware resources are continuing to be occupied and not deoccupied. As the need for more processing to execute there will also be need for more resources, if there are now free resources then the processes will have nowhere to go which can cause a decline in system performance. This is why it is important to deallocate semaphores similarly to the important of deallocating memory. To combat this I believe that the operating system should implement a garbage collector similar to that of a compiler to avoid these issues. I would have the garbage collector run periodically to free up any resources not freed by the user.

Summary:

This assignment gave me a dive into a topic in computer science that I was unfamiliar with. I was unaware about semaphore and their contribution to operating systems. What I took away from this assignment is just how important semaphores are in concurrent programming. Semaphores are great for managing resources, which is important when resources are limited or system performance is essential. They are also great for synchronization of process or threads. They also help in avoiding any deadlocks that may commonly occur. Although there are many other things that semaphores solve, these are the key points that stood out to me during this assignment.

I find it interesting that there is so much more going on within the hardware of a computer and the software controlling it than I originally thought. This course has revealed concepts about computing that I never even heard about let alone knew about. The more I went through all the assignments and the course the more I was intrigued with computers on a lower level. I originally wanted to focus on the hardware and low level computing for my career and this course confirmed that this is the field I would like to delve into a bit more. Thank you for a great semester and the knowledge that was given to us by you Dr. Moore and the help that you gave Ariath. I hope you both continue to find success in your careers and lives!

RESOURCES

- The C Programming Language, 2nd edition, by Brian Kernighan and Dennis Ritchie
- xv6: a simple, Unix-like teaching operating system, by Russ Cox, Frans Kaashoek, and Robert Morris. RISC-V version.
<https://pdos.csail.mit.edu/6.828/2021/xv6/book-riscv-rev2.html>
- Operating Systems: Three Easy Pieces, by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau.
<https://pages.cs.wisc.edu/~remzi/OSTEP/>