Project 2: Semaphores

Jonathan Poisson – Poisson.Jonathan777@gmail.com

To get the patch to run, I had to do the following. This will assure the server starts up

1) cd /usr/src
2) run `git apply --reject --whitespace=fix (path to project2-submission.patch)`
3) make build && make install && reboot

Overall Implementation

Server

I decided to created support for semaphores in Minix 3.2.1 by creating a standalone semaphore service. Instead of adding semaphore operations to the PM or VFS server, I found that I had more freedom and ease of implementation by creating a new service to run a unique server. This service controls the creation of semaphores, the waiting queue, and the up and down operations on the semaphores. The service is started up automatically when the OS boots by adding the service to `/kernel/table.c`.

Below is a brief explanation of the operations. All of the semaphore data and semaphore array manipulation operations is held in semaphore_data.c and the syscalls that control these operations is held in semaphore_operations.c I tried to treat is almost like a model, view and controller pattern (MVC with the view being the user calls).

Note: Because Dr. Newman told me that the semaphore number should be dynamic, I originally implemented the semaphores as a linked list allowing for dynamic construction and unlimited semaphores. However, after speaking with David in discussion on April 4th, I changed the semaphore to be a fixed-sized array but left the process queues as a linked list so all my work creating a linked list class didn't go to waste.

For a shorter version of what is below, I have created man pages that show up when running either `man create_semaphore`, `man delete_semaphore`, `man up`, or `man down`.

Create Semaphore

Create semaphore takes the initial value and type of semaphore to create the semaphore in the semaphore service. Internally, there is an array of semaphores (allowing for up to 99 unique semaphores) and the create_semaphore operation is successful if there is an open slot in the semaphore array, allowing the slot to be filled with the semaphore value. The semaphores are created in a semaphore_node structure that its type, value, who created it, and a pointer to its waiting process queue. The return value is a semaphore handle that represents the index in the semaphore array where the created semaphore resides.

Delete Semaphore

Delete semaphore takes a semaphore handle (index in semaphore array) and has the purpose of freeing a spot in the semaphore array (setting its value back to -1 and all other fields to NULL). The call is successful and will return -1 if the semaphore handle is a valid entry in the array and has a created semaphore in the given index. The call to delete_semaphore can will fail if there are any processes waiting in the process queue, however.

Down

Down takes a semaphore handle, and the endpoint of the process (The user does not have to worry about this, it is handled in the semaphore server's main.c). If the semaphore handle passed by the calling process is a valid one, the value of the semaphore is checked. If the value is 0, the process is added to the waitingProcesses queue and is suspended, if not, the semaphore value is decremented by 1 and the process continues running. The waiting process queue is a linked list like implementation. Each waiting process is pointing to the next waiting process, and new process are added onto the end of the queue. If, for some reason, the process queue tries to wake up a waiting process and the process happens to have died, it will continue to wake up the next waiting process until it is successful (this will avoid deadlock).

Up

Up takes a semaphore handle (index in semaphore array) and has the purpose of incrementing the current semaphore. If the specified semaphore is of type binary and the caller wants to up a 1, it **will fail**. If the current value of the semaphore is 0, the up operation will first check to see if there are any waiting processes in the waiting processes array. If there is one, it will be woken up with a message, removed from the waiting process queue, and the semaphore value will remain the same.

Bugs

Server permissions

When I created the server, I had issues getting processes to have access to the server. The server could also not send and receive messages, even with the necessary access in `/usr/src/etc/system.conf` of...

```
service semaphore
{
    uid 0;
    ipc ALL;
    system ALL;
}
```

To get around this, I added an exception in `/usr/src/kernel/proc.c` around linke 473. The exception looks for endpoint 11 (the endpoint I used for the semaphore server) as the source or destination of the message, and allows the message to pass through without error. I would rather figure out how to set the permissions correctly, but after hours of googling and messing with different files I had no luck. If the user wants to refresh the service, they will need to reset the system, as refreshing the service is also blocked.

Garbage collection

Because my implementation of semaphores was a separate server, I had some issues linking processes to the created semaphores. I exhausted every solution I could think of to fix the last test case (cleanup) where a process creates semaphores until there is no space, and then the semaphore is deleted when the process ends. I had issues sharing data between the PM server and my semaphore server. Even with a global array and explicit messaging between each server, the OS would either have two separate copies of the global array or would crash due to deadlock with too many messages being passed. My solution is as follows...

Each semaphore created stores the endpoint of the process that created it. I used endpoints because they are more easily available in the semaphore server as the server is woken up with messages anyways. If a process tries to create a semaphore but the semaphore array is full, garbage collection will occur. The garbage collection will delete the *ONLY* the semaphores that have an endpoint that matches the process calling create_semaphore. This solves the test case that was sent out by the TA, as an individual process is in a loop creating useless semaphore. The bug is that if the user were to run 99 unique processes and not delete any of the semaphores they created, no new semaphores would be allowed. However, if the programmer makes sure to delete the semaphore that the process created at the end of the code (or even runs another program that deletes any semaphore from 1-99), the semaphore service will oblige and there will be no issues.