CIS 452 – Operating Systems

Project 1 – One Bad Apple

February 24th, 2023

Evan Johns, Alexis Webster

INTRODUCTION

For this project, we were tasked with creating a circular communication system with a dynamic length. The system would consist of multiple nodes, each of which could only read from the node before it, and write to the node in front of it.

The number of nodes in the system could not be statically defined, but rather had to be defined by the user when the program started. Each node in the system had to be an individual process, and would communicate with the node behind it and the node in front of it through the use of pipes.

The user would prompt a message and to which node the message should be sent to, and the nodes would be responsible for sending that message to the correct node where it would be printed. The program had to also handle receiving an interrupt signal, where all processes would be properly shut down and the program would exit.

There are many different approaches that one can take when trying to implement a system that follows these requirements. In the following sections, we will explain how we approached this problem and the obstacles we encountered in the process.

DESIGN

As for the design of the project, we used pipes and forked processes to achieve the results for the project. The program asks the user for the number of nodes they desire, and then it uses that value to create an array of pipes. This array of pipes is a 2-dimensional array, where the first index of the array represents the pipe’s index, and the second index of the array represents the file descriptors (read and write) for that pipe.

Next, the program initiates the node creation process. Using a for loop, the program creates a new child process and assigns the *in* and *out* variables of each process to their corresponding pipe *read* and *write* file descriptors. The pipes between each node are created in such a fashion that the first node’s *in* needs to match the pipe for the last node’s *out*. This allows the communication between the nodes to be circular.

At this point in the program, the child processes (nodes) are waiting to read some input from the main process, and the main process is waiting for the user to input a message. The main process takes in the message and the node that the user wants to send it to, and then the main process wraps the message in a packet structure which contains a *data* field and a *dstNum* field. This packet is sent to the first node in the ring using write() and the main process’ *out* file descriptor (this has been set to the same *out* as the last node’s *out* in the child creation process).

Next, each node reads the packet’s *data* and *dstNum* using read(). The node then evaluates the *dstNum* field and determines if the message is intended for them. If it was, then the node prints out the message, and does not pass the packet onto the next node. However, if the message was not intended for that node, the packet is sent off to the next node using write() and the *out* variable (such that the *out* variable corresponds to the pipe the next node’s *in* is connected to).

The program also has error checking throughout it, all of them making sure that the system calls used, such as fork(), read(), and write(), return a value greater than or equal to 0, which indicates that there was no error. If an error does occur, a descriptive error message is displayed to the console.

The program also implements handling for the SIGINT signal at the beginning, along with a signal handler method, in order to exit the program gracefully with control-c.

OBSTACLES

Throughout the development process, we ran into quite a number of issues. One issue in particular that we ran into had to do with accessing pipes. In the main process, the *in* and *out* file descriptors are defined for each child process during that given process’ initialization. The actual creation of each pipe was declared above the child process creation portion of the code, and each pipe was put into a pipes array. We started running into issues when we found that user messages were not being sent through the correct pipes. After much deliberation, we found that we had accidentally reversed our pipes’ *in* and *out* ends when they were being generated; We used fd[0] for write and fd[1] for read by accident. Once we discovered the issue, it was easily fixed by swapping the numbers in each pipe’s creation, and the program functioned as intended.

Another obstacle we ran into was that the logic for reading and writing to the pipes (line 91) for each child process was not functioning as intended. After reading the man pages and various support topics online, we knew that the *read()* system call would wait on a pipe until new information had been written to the pipe, so we were confused when the loop that was calling read in the child processes was resulting in some unexpected behavior. We spent quite a bit of time troubleshooting and trying to figure out what the problem was. It wasn’t until, in a last ditch attempt, we changed the *while* loop to a *do-while* loop that the child processes read and wrote to the pipes as intended. We tried researching why this would be causing an issue, but unfortunately, we were never able to figure out why the *while* loop would have different behavior to a *do-while* loop.