OS Assignment 3 - Report for Q2

By: Arnav Goel (2021519), Section B, Branch: CSAI

In this question, we are asked to generate 50 random strings, which we then send from our process P1 to another Process P2 in batches of 5 and then receive the highest ID back from P2 to print it in P1. For this, we employ 3 IPC techniques:

I) FIFOs

For the first IPC mode, we use FIFOs or Named Pipes for sending strings in batches of 5. For this, we create a FIFO called "myfifo" using mkfifo(), giving it the necessary permissions. This first calls the **open()** call on this FIFO to get its file descriptor and then uses it to call the **write()** syscall to write 5 strings to process P2. Process P2 then calls the **read(),** which reads the strings from this FIFO. We also create another FIFO "myfifo2" using **mkfifo()** in Process P1. Process P2 **opens() myfifo2** and uses **write()** to write the highest index in **myfifo2,** which is then read by Process P1 using the **read()** call. I also use the **usleep() function** to ensure some delay in sending the data from my client to the server. This helps prevent any unexpected output or deadlock situations on our FIFO.
Note: This needs to be run using two terminal windows.

II) Shared Memory:

We use shared memory in our 2nd part, wherein we create 5 unique keys corresponding to 5 keys and 5 unique pointers to 5 shared memory segments. So when we store any batch of 5 strings in these 5 memory segments in the **sender.c** code and then receive it at the **receiver.c.** After printing these strings, we send the highest ID back and destroy these shared memory segments in the receiver process. **In the loop, while sending our strings,** we use **shmget() to create a particular memory segment corresponding to one key.** Then we use **shmat() to attach this created shared memory segment to our process and copy our string to this segment.** We then detach it from process **sender.c,** and we do this 5 times before calling fork() to execute the process **receiver.c** using the execl() call. These segments are then deleted by the **receiver process,** which reads these strings from the shared memory segment before deleting them. Thus in our code, **we call fork() 10 times** as we only need one call to **fork() to** send and receive a batch of 5 strings.

III) UNIX Domain Sockets:

Our socket_receiver.c code acts as the **Server,** which receives strings from socket_sender.c, which acts as the **Client.** We used the AFX_UNIX family and SOCK_STREAM protocol to create our UNIX sockets. The server first creates using the

socket() call and then calls **bind() to** bind to the stream. We then call the **listen()** call to queue incoming connections to the server. Then we call the **accept()** call to accept a connection. We run the server before, and the server reaches the **accept()** call and hangs there till we send a connection through the **client.** Now in the **client.c** we create the same socket and connect it using the **connect() call.** We then write 5 strings to our server using the **write()** call, which is then read by the server one after another by the **read() call. This** while nested inside the while loop ends only when our total IDs sent number equals 50, and so does at the server end. When it reaches 50, the server shuts down the accept() call and closes the socket from its end. When **the highest ID received by the client back** is 50, we close the socket on the client end to end our IPC.

NOTE: In all three IPC modes, we have implemented clock_gettime based on assignment 2 to clock the time taken by the client side on all the parts for conducting the entire IPC procedure.