Q2 Write-Up

Unix domain sockets IPC (flow-controlled)

Working of the Program(s)

For FIFO Programs:

For these programs, I have utilized a structure named ‘packet’, that stores the ID and contents of a string that is to be passed. Each time, 5 such packets are sent to the Program P2 from Program P1.

First, I have generated 50 random strings of length 9 in program P1 using loops appropriately. I have created a buffer array of packets as buffer[5] that we will use to send 5 such packets. For the initial message, the first five strings (IDs 0-4) are copied onto this buffer packet.

Now, I have created a named pipe(FIFO) using mkfifo() instruction. We then open this named pipe using open() system call and write the contents of the buffer packet array using the write() system call and close the file descriptor using close(). This essentially sends the data through the pipe. To acknowledge the message received from Program P2, we open the pipe once again using open() and read the message (i.e the highest ID received) from P2 and store it in an integer named ‘idbuffer’, and then close(). Depending on this idbuffer, the next set of 5 packets are sent to P2 in a similar fashion and the program ends in case the ID is 49 (the last ID).

In Program P2, we use the same ‘packet’ structure to receive messages. P2 opens the same named pipe and reads the content and stores it into its own buffer[5] array of packet structure. Then, P2 checks for the highest ID among the strings received and sets an integer variable ‘id’ to this highest ID. To send this ID, P2 opens the pipe and writes this ‘id’ integer to it so that P1 can acknowledge and receive it.

For Message Queues:

In these programs, I have utilized two structures msg_st and msg_highid. The former is used to send the 5 packets as described above to P1 and the latter is used to send the highest ID among the 5 strings back to P2. Each of them also have an mtype indicating the type of message. I have used mtype ‘1’ for msg_st and mtype ‘2’ for msg_highid.

In P1, first we generate an IPC key using ftok() that is used to convert a pathname and a project identifier to an IPC key and use msgget() to get the message queue identifier. We generate 50 random strings in a similar fashion described in FIFOs and use the initial 5 IDs (0-4) as the first strings to be passed. After that, we use msgsnd() to send the initial 5 strings and their IDs as a struct (stored in ‘message’ variable of type msg_st). We utilize the same message queue to receive the acknowledgement from program P2 and use msgrcv() to receive the message (i.e the highest ID) from P2 by specifying the mtype as ‘2’ while receiving so that messages of type ‘msg_highid’ are received from the queue. We store this message in the ‘idbuffer’ variable (which is of type ‘msg_highid’) so that P1 can access the received ID. Depending on this idbuffer, the next set of 5 packets are sent to P2 in a similar fashion and the program ends in case the ID is 49 (the last ID).

In P2, we generate the same IPC key in a similar manner as described above and get the message queue identifier. Here, we use msgrcv() with mtype ‘1’ to receive the message from P1 and store it in the ‘message’ variable of type ‘msg_st’. Now, the 5 strings and their IDs can be accessed by P2. P2 then finds the highest ID among the strings and sends it back to P1 using the same message queue using the ‘idbuffer’ variable of type ‘msg_highid’ so that P1 can acknowledge this.

For Domain Sockets:

For these programs, I have utilized a structure named ‘packet’, that stores the ID and contents of a string that is to be passed. Each time, 5 such packets are sent to the Program P2 from Program P1. I have created a buffer array of type ‘packet’ as buffer[5] that we will use to send 5 such packets. For this implementation, P2 acts as the server and P1 as a client. I have also used the ‘sockaddr_un’ struct for the remote and local connections.

In P2, we use the socket() call to make a UNIX domain socket and store the return value in variable ‘s’ which is the socket descriptor. Then we bind this to an address in the UNIX domain using bind(). After that we call listen(), that makes the socket to listen for any incoming connections from clients, with 5 incomming connections that can be queued. After that in the loop, we accept() connections from clients (i.e P1) which returns another socket descriptor which is connected to the client. After acception, we call recv() that receives the data packets from P1 and stores it in the ‘buffer’ array. Then, P2 checks for the highest ID among the strings received in the buffer and sets an integer variable ‘id’ to this highest ID. After that, we use the send() call on the same socket descriptor to send this ID back to P1 for acknowledgement and close() this socket descriptor.

In P1, I have generated 50 random strings of length 9 as described above for FIFOs. The buffer array of packets as buffer[5] will be used to send 5 such packets. For the initial message, the first five strings (IDs 0-4) are copied onto this buffer packet. After this, we call socket() to get the UNIX domain socket to communicate through and set up the sockaddr_un struct with the address of remote. Then we call connect() with this as an argument to establish a connection. Once connected, P1 can use send() and recv() to send and receive data packets respectively. The contents of the ‘buffer’ packet array are sent to P2 using send() and the highest ID is received as acknowledgement from P2 and stored in the integer variable ‘idbuffer’. Depending on this idbuffer, the next set of 5 packets are sent to P2 in a similar fashion and the program ends in case the ID is 49 (the last ID).