Name: Chaitanya Arora
Roll No. 2021033

Readme File for Question 2 Assignment 3
Write two programs P1 and P2. The first program P1 needs to generate an array of 50 random strings (of characters) of fixed length each. P1 then sends a group of five consecutive elements of the array of strings to P2 along with the ID's of the strings, where the ID is the index of the array corresponding to the string. The second program P2 needs to accept the received strings, and send back the highest ID received back to P1 to acknowledge the strings received. The program P2 simply prints the ID's and the strings on the console. On receiving the acknowledged packet, P1 sends the next five strings, with the string elements starting from the successor of the acknowledged ID.

In fifo:

Just use the makefile to run the program and the output would be printed on the terminal.

This code is using named pipes (also known as FIFOs) to communicate between two processes. Named pipes are a way for processes to communicate with each other by reading and writing to a special file on the file system, instead of using traditional inter-process communication (IPC) mechanisms such as pipes or sockets.
The #define FIFO_NAME "one" and #define FIFO_NAME2 "two" statements define constants for the names of the two FIFOs that will be used for communication. The char strings[6 * 50] array is an array of characters that stores a string of characters that will be sent between the processes. The int num, fd, and fd2 variables are used to store the number of bytes read or written, and the file descriptor for each FIFO. The clock_t begin variable is used to store the start time of the program. The double time_spent variable is used to store the elapsed time of the program.
The receive_index() function is called by the func() function to receive the index of the next chunk of the string that the func() function should send. It opens the second FIFO (FIFO_NAME2) in read-only mode and reads a single integer from it. It then calls the func() function with the received index, and closes the FIFO.
The func() function is the main function that sends chunks of the string to the other process and receives the index of the next chunk to send. It takes an integer parameter (res) which is the index of the first character in the chunk of the string to send. If the index is greater than 279 (which is the last index of the string), it prints a message and returns 0. Otherwise, it opens the

first FIFO (FIFO_NAME) in write-only mode, creates a new string (s) with the next 30 characters from the input string, and writes the new string to the FIFO. It then closes the FIFO and calls the receive_index() function to receive the next index to send.
In the main() function, the program initializes the random number generator, creates the FIFOs, generates a random string of characters and stores it in the strings array, and stores the start time of the program in the begin variable. It then calls the func() function with an initial index of 0 to start the communication. Finally, it returns 0 to end the program.

In the receiving program
This code is using named pipes (also known as FIFOs) to communicate between two processes. Named pipes are a way for processes to communicate with each other by reading and writing to a special file on the file system, instead of using traditional inter-process communication (IPC) mechanisms such as pipes or sockets.
The #define FIFO_NAME "one" and #define FIFO_NAME2 "two" statements define constants for the names of the two FIFOs that will be used for communication. The int num, fd, and fd2 variables are used to store the number of bytes read or written, and the file descriptor for each FIFO.
The send_index(int index) function is called by the func() function to send the index of the next chunk of the string that the other process should send. It opens the second FIFO (FIFO_NAME2) in write-only mode and writes a single integer (the index) to it. It then closes the FIFO and calls the func() function to receive the next chunk of the string.
The func() function is the main function that receives chunks of the string from the other process and sends the index of the next chunk to receive. It opens the first FIFO (FIFO_NAME) in read-only mode and reads up to 30 characters from it into a string (s). It then searches the string for the character with the highest ASCII value (which represents the index of the next chunk to receive) and stores the index in the max_index and index variables. It then prints the character and the 5 characters after it, and closes the FIFO. Finally, it calls the send_index() function with the index of the next chunk to receive.
In the main() function, the program creates the FIFOs and calls the func() function to start the communication. Finally, it returns 0 to end the program.

I have used `fd2 = open(FIFO_NAME2, O_RDONLY);`
And this will help me in opening one end of the pipe for reading
I have read the contents of the pipe using num = read(fd2, c, sizeof(c)
Then i have used `fd = open(FIFO_NAME, O_WRONLY);`
And this will help me in opening one end of the pipe for writing data.
I am using mknod(FIFO_NAME, S_IFIFO | 0666, 0);
For making a pipe.

I am using   close(fd);
For closing the pipes that i have used for IPC.

In socket:

Just compile the program using the make file and then the last line of the output would be the total time taken to run acknowledge the 50 strings. Then all the data is stored int he ipc_logs.txt file which is also turned in.

This is a program that performs interprocess communication (IPC) using a Unix domain socket and the fork() system call to create a child process. The program generates a string of 300 characters, with each block of 6 characters starting with a unique uppercase letter from A to Z. The parent process sends 30 characters at a time to the child process through the socket and waits for a response. The child process receives the 30 characters, identifies the block with the highest starting letter, and returns the first character of that block to the parent process. The parent process uses the returned character to determine the starting index for the next block of 30 characters to send to the child process. This process is repeated 10 times.
The program also uses a file ipc_logs.txt to log the input strings sent to the child process and the maximum index and string identified by the child process. The program also calculates and prints the elapsed time for the IPC process.
The program uses the following variables:
counter: an integer variable that is used to keep track of the number of times the IPC process has been repeated. It is initialized to 0.
begin: a clock_t variable that is used to store the starting time of the IPC process. It is initialized with the current time using the clock() function.
end: a clock_t variable that is used to store the ending time of the IPC process. It is initialized with the current time using the clock() function.
time_spent: a double variable that is used to store the elapsed time for the IPC process. It is initialized to 0.0.
strings: a char array that stores the generated string of 300 characters.
num: an integer variable that is used to store the number of bytes read or written in the socket.
fd: an integer variable that is used to store the file descriptor for the socket.
fd2: an integer variable that is used to store the file descriptor for the ipc_logs.txt file.

sv: an integer array of size 2 that is used to store the file descriptors for the socketpair created using the socketpair() function.
buf: a char variable that is used to store the first character of the block identified by the child process.
The program has the following functions:
main(): the main function of the program. It generates the string of 300 characters and initializes the starting index for the block of characters to send to the child process. It then enters a loop that is repeated 10 times, in which it sends 30 characters to the child process and receives the first character of the identified block. It uses this character to determine the starting index for the next block of characters to send to the child process. The function also calculates and prints the elapsed time for the IPC process and logs the input strings sent to the child process and the maximum index and string identified by the child process in the ipc_logs.txt file.

I have used socketpair(AF_UNIX, SOCK_STREAM, 0, sv)
To use the terminal in socket data stream mode and then, i am creating a child program and in I am sending randomly generated strings to that program and then in the child programi am sending back the id of the max string and then this process is repeated untill all the string have been sent to the child program.

In shared memory:

The code initializes two semaphores, sem and sem2, using the sem_init() function. The first argument to sem_init() is a pointer to the semaphore, and the second argument is the number of processes that can access the semaphore simultaneously. The third argument is the initial value of the semaphore.
The sem semaphore is used to synchronize access to the shared memory region. Before a process can access the shared memory, it must wait on the sem semaphore using the sem_wait() function. This causes the process to block until the semaphore is signaled, at which point the process can proceed.
The sem2 semaphore is used to synchronize access to the time_spent variable. Before a process can update the time_spent variable, it must wait on the sem2 semaphore using the sem_wait() function. This ensures that only one process can update the time_spent variable at a time.
The sem_post() function is used to signal a semaphore, allowing a waiting process to proceed. In this code, the sem semaphore is signaled after the shared memory has been updated, and the sem2 semaphore is signaled before the time_spent variable is updated.
Overall, the semaphores in this code are used to synchronize access to shared resources and ensure that multiple processes do not try to access the same resource simultaneously.

This C program uses interprocess communication (IPC) to write and read data to and from a shared memory segment. The shared memory segment is created using the shmget function and attached to the process's address space using the shmat function.
The program also uses semaphores for synchronization. Semaphores are a type of IPC mechanism used to control access to shared resources. In this program, two semaphores are created using the sem_init function and initialized with a value of 1. The semaphores are used to ensure that only one process can access the shared memory at a time.
The program begins by defining some constants and variables. SHM_SIZE is defined as 4096, and time_spent, strings, num, fd, and fd2 are all variables of different types.
The main function then generates a key using the ftok function and the path to a temporary file and the character 'A'. The key is used to create a new shared memory segment using the shmget function, with a size of SHM_SIZE and permissions of IPC_CREAT | 0666. If the shmget function fails, it prints an error message and returns 1.
The shared memory segment is then attached to the process's address space using the shmat function. If this function fails, it prints an error message and returns 1.
Two semaphores are then created using the sem_init function and initialized with a value of 1. If either of these function calls fail, it prints an error message and returns 1.
The program then generates random characters and stores them in the strings array. It then enters a loop that will run 10 times.
Inside the loop, the program waits for the first semaphore using the sem_wait function. It then waits for the second semaphore using the sem_wait function.
The program then reads data from the shared memory segment using a pointer to the start of the segment, which is located at shm_ptr + sizeof(sem_t). It prints this data to the console using the printf function.
The program then writes new data to the shared memory segment using the memcpy function, copying a string containing the loop iteration number to the segment.
The program then releases the first semaphore using the sem_post function and sleeps for 0 seconds. It then releases the second semaphore using the sem_post function and sleeps for 0 seconds.
Finally, the main function returns 0 to indicate success.