Design Document:

Sender file:

This program shall implement the process that sends files to the receiver process.
It shall perform the following sequence of steps:

1. The sender shall be invoked as ./sender file.txt where sender is the name of the executable and file.txt is the name of the file to transfer.
2. The program shall then attach to the shared memory segment, and connect to the message queue both previously set up by the receiver.
3. Read a predefined number of bytes from the specified file, and store these bytes in the chunk of shared memory.
4. Send a message to the receiver (using a message queue). The message shall contain a field called size indicating how many bytes were read from the file.
5. Wait on the message queue to receive a message from the receiver confirming successful reception and saving of data to the file by the receiver.
6. Go back to step 3. Repeat until the whole file has been read.
7. When the end of the file is reached, send a message to the receiver with the size field set to 0. This will signal to the receiver that the sender will send no more.
8. Close the file, detach shared memory, and exit.

Receiver file:

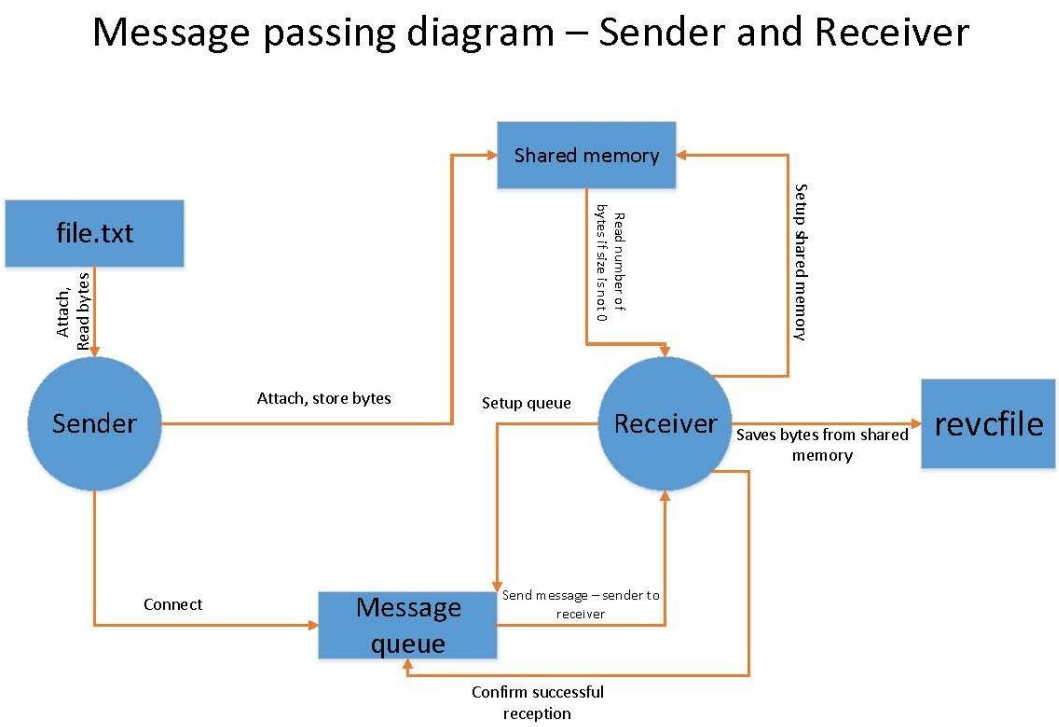
This program shall implement the process that receives files from the sender process. It shall perform the following sequence of steps:

1. The program shall be invoked as ./recv where recv is the name of the executable.
2. The program shall setup a chunk of shared memory and a message queue.
3. The program shall wait on a message queue to receive a message from the sender program. When the message is received, the message shall contain a field called size denoting the number of bytes the sender has saved in the shared memory chunk.
4. If size is not 0, then the receiver reads size number of bytes from shared memory, saves them to the file (always called recvfile), sends message to the sender acknowl- edging successful reception and saving of data, and finally goes back to step 3.
5. Otherwise, if size field is 0, then the program closes the file, detaches the shared memory, deallocates shared memory and message queues, and exits.

Goal of our project:

To successfully transfer filecontents of ‘keyfile.txt’ to ‘recvfile’.

Flow Chart:



Pseudocode for Sender file:

// size of shared memory chunk
SHARED_MEMORY_CHUNK_SIZE 1000

// id for shared memory segment and message queue
shmid, msquid

// pointer to shared memory
sharedMemPtr

```
/*
        Sets up shared memory segment and message queue
        @param shmid - id of the allocated shared memory
        @param msqid - id of shared memory
*/
init function(shmid, msqid, sharedMemPtr) {

        // generate a key and save it to a variable
        key = ftok(keyfile.txt, a)

        // save the id of the shared memory segment
        shmid = shmget(key, SHARED_MEMORY_CHUNK_SIZE, some shmflag)

        //check to see if the size of the shared memory segment is correct
        if(shmid is -1)
        {
                print “creation of shared memory segment id was unsuccessful”
                exit(1)
        }

        // attach id to the shared memory
        sharedMemPtr = shmat(shmid, some shmaddress, some shmflag)

        // check to see if attachment to shared memory was successful
        if sharedMemPtr == -1
                print “attachment to shared memory was unsuccessful)
                exit(1)

        // attach id to the memory queue
        msqid = msgget(key, some msgflag)

        // check to see if the message queue was successfully attached
        if msqid == -1
                print “ attachment to the message queue was unsuccessful”

        /* store the IDs and the pointer to the shared memory region
        in the corresponding parameters */

}

/**
 * Performs the cleanup function
```

```
 * @param sharedMemPtr
 * @param shmid - the id of the shared memory segment
* @param msqid - the id of the message queue
*/

void cleanUp(shmid, msqid, sharedMemPtr) {
        shmdt(sharedMemPtr)
        print “detaching from memory segment”
}

/* Detach from shareMemptr */
        void* sharedMemPtr

        shmdt(sharedMemPtr)
         print: "Detaching from memory"

send(filename) {

        // send message to receiver to signal data is ready
        // use signal() and wait()?
        signal(sndMsg)

        /* wait until receiver sends message of type RECV_DONE_TYPE.
                Tells us that receiver finished saving the memory chunk
        */
        wait(rcvMsg)

        ****************************************************************
        /*
            Tell the receiver that we are finished sending messages.
             Send a message (of type SENDER_DATA_TYPE) with size field set to 0
        /*
        sndMsg = 0
        signal(sndMSG)

}

   /* Let the receiver know that the data is ready to be sent
   Message type is SENDER_DATA_TYPE */

        print out :" Sending data to receiver"
        sndMsg.type = SENDER_DATA_TYPE;
```

```
/*Make an if statement to catch any errors if the data could not send */

        if statement: (retrieves anything that is less than 0
        stating that was an error)
         print "error"

/* If the if statement is FALSE */

        print : "Message success"


/* Wait for the receiver the receive all the data */

         print: "Receiver is receiving the data"

/* Using a do-While loop, get the message from the queue
 and place into rcvMsg */

         do ( msgrcv( retrieve data using msqid)
        while( rcvMsg.mtype != RECV_DONE_TYPE);


 /* Set the message type to equal SENDER_DATA_TYPE
  The size of it should be 0 */

         sndMsg.mtype = SENDER_DATA_TYPE;
         sndMsg.size = 0;


/* Make an if statement to catch any errors */

         if statement: (retrieves anything that is less than 0
        stating that was an error)
        print "error"


 /* State the program has completed */

         print : " Sent Program Completed"
```

```
main(argc, argv) {
        /* check the command line arguments */
        if(argc less than 2) {
                print “USAGE: <FILE NAME>”
        }

        /* connect to shared memory and the message queue */
        init(shmid, msqid, sharedMemPtr)

        /* send the file */
        send(argv[1])

        /* cleanup */
        cleanUp(shmid, msqid, sharedMemPtr)

        return 0
}
```

****

Pseudocode for Receiver file:

```
int shmid, msqid //id for shared memory and message queue
void sharedMemPtr
char recvFileName

//Initialize the shared memory segment and message queue
init(shmid, msqid, sharedMemPtr)
{
        // generate a key and save it to a variable
        key = ftok(keyfile.txt, a)

        //get the shared memory
        shmid = shmget(key, SHARED_MEMORY_CHUNK_SIZE, some shmflag)
        //attach shared memory
        sharedMemPtr = shmat(shmid, null, 0)
        //make message queue
        Msqid = msgget(key, some flag)
```

```
}

mainLoop()
{
	Msgsize = 0

	//open the file
	Fp = fopen(recvFileName, write);


	If (file is not open)
	{
		Throw error
		Exit program
	}


Message temp

//Receive a message and put it into temp, then check for error

if(fail to receive message)
{
	Throw error
}

//Get new message size
msgSize = temp.size

while(msgSize is not 0)
{
	//If sender isn't telling us we're done, keep going
	if(msgSize != 0)
	{
		Print "write to file"
		//Save memory to file
		if (fail to save memory) {
			Throw error
		}

	//Let sender know we are ready for next file chunk
	Message doneType
	doneType.mtype = RECV_DONE_TYPE
```

```
        doneType.size = 0

        //Send the message
        if(sent message is < 0)
        {
                Throw error
        }
        Print “We are ready for the next chunk”

        //Get next message
        if(fail to receive message)
        {
                Throw error
        }

        //Set the msgSize to the new message size where temp is the new message
        msgSize = temp.size
}
Else
{
        Close file
}


cleanUp(shmid, msqid, sharedMemPtr)
{
        //Detach from shared memory
        Print “Detaching from shared memory”
        shmdt(sharedMemPtr)
        Print “Deallocating shared memory chunk”
        shmctl(shmid, ipc rmid , null)
        Print “Deallocate message queue”
        msgctl(msqid, ipc rmid, null)
        Print “Cleanup is done”
}

ctrlCSignal(int signal)
{
        //Free system resources
        cleanUp(shmid, msqid, sharedMemPtr)
        Exit
}
```

```
main(argc, arvg) {

        //Get signal handler so ctrl-c will delete message queues and shared memory before
exiting. This is used in ctrlCSignal

        signal(SIGINT, ctrlCSignal)

        //Initialize the shared memory id, message queue id, shared memory pointer
        init(shmid, msqid, sharedMemPtr)

        //start main loop
        mainLoop()

        //deallocate shared memory, message queue, and detach from shared memory
        cleanUp(shmid, msqid, sharedMemPtr)

        Return 0
}
```