# **State University of New York, Albany**

A purple and yellow logo

Description automatically generated

**Assignment - 3**

**ICSI 500**

**DEPARTMENT OF COMPUTER SCIENCE**

**Submitted To:**

Professor. JACKSON MARQUES DE CARVALHO

**Submitted By:**

TYLER DAVIS

**Email:** tgdavis@albany.edu

# **Table of Contents**

|  |  |  |
| --- | --- | --- |
| **S No.** | **Contents** | **Page No.** |
| 1 | System Documentation:   1. Data flow Diagram 2. Routines and Description 3. Implementation Details |  |
| **2** | Test Documentation |  |
| **3** | User Documentation |  |

# **System Documentation**

**Data Flow diagram:**

A diagram of a project

Description automatically generated with medium confidence

**Routines:**

**Producer(Child Process):**

* Reads “filename.inpf” and writes it to a buffer
* Modifies the contents by encoding, adding parity bits, and framing the binary data
* Writes the binary contents to a file
* Uses writePipe() to send the frames to the Consumer
* Gets frames from the parent using readPipe()
* Modifies the contents by deframing, removing paridy bits, and decoding the data
* writes the final message to a file

**Consumer(Parent Process):**

* Gets the file content from Producer process using readPipe() on the first pipe
* Modifies the contents by deframing, removing paridy bits, and decoding the data
* Uses toUpperCase() to return the lower case version of editSource.txt and writes the output to upperCase.txt
* Modifies the contents by encoding, adding parity bits, and framing the binary data
* Writes the binary contents to a file
* Sends the file names back to the Consumer through the second pipe using writePipe()

1. ssize\_t writePipe(int fd, const void\* buf, size\_t count):

* Writes the data from a buffer to a pipe
* *fd* is the file descriptor of the pipe to write to
* *buf* is the data being written to the pipe
* *count* is the size of the data being written to the pipe

1. ssize\_t readPipe(int fd, void\* buf, size\_t count):

* Reads data from a pipe to a buffer
* *fd* is the file descriptor of the pipe to read from
* *buf* is the median to which the data being read from the pipe will be stored
* *count* is the size of the buffer

1. char\* encodeMessage(const char\* inputbuffer)

* converts the input string from ASCII characters to binary and returns it
* *input buffer* is the input string in ASCII

1. char\* decodeMessage(const char\* inputbuffer)

* converts the input string from binary back to ASCII characters and returns it
* *input buffer* is the input string in binary

1. char\* addParityBits(char\* bitArray)

* adds the parity bits to the given input binary string and returns the new value
* *bitArray* is the input bit array

1. char\* removeParityBits(char\* bitArray)

* removes the parity bits to the given input binary string and returns the new value
* *bitArray* is the input bit array

1. char\*\* convertToFrames(char\* bitArray)

* converts the given bit array to frames and returns an array consisting of those frames
* *bitArray* is the input binary string

1. char\*\* unconvertTheFrames(char\* frameArray[100][255])

* unconverts the given array of frames to a binary string and returns that string
* *frameArray* is the inputted array of frames

1. int toUpperCase(char\* input):

* Converts all lowercase characters to upper case characters in a file
* *Input* is the file name passed in to be read

# **IMPLEMENTATION DETAILS**

Assignment3.c implements a full duplex between a parent and child process. The childs reads a text file (filename.inpf) and collects that information in a buffer. It then encodes that data, puts parity bits on that data, and frames the data before sending it through the first pipe to the parent. The parent then receives the data through the first pipe. It then deframes, removes the parity bits, and decodes the data. It then runs an uppercase function on that data and takes the same encoding steps as the child before feeding it through the second pipe. The child then reads the data from the second pipe, takes the necessary decoding steps (which should retrieve the uppercase version of the original text) and then writes the data that it got the file.

This file also contains the implemented methods whose declaration are present in “encDec.h” which implements all necessary process for encoding/decoding, handling parity bits, and framing.

**Data Flow:**

* **Child Process** reads filename.inpf. It then encodes, adds parity bits, and frames that data before sending it to the parent process through the first pipe
* **Parent Process** Receives the data from the child and takes the necessary steps to decode, remove the parity bits, and deframe the content that it received. It then uses an uppercase function by calling an exec() program and reencodes, readds the parity bits, and reframes the data before sending it through the second pipe back to the child
* **Child Process** Receives the frames through the second pipe, then takes the necessary steps to decode, remove the parity bits, and deframe the data before reading the modified uppercase text. It then writes the data it got to a file

# **User Documentation**

This program simulates a full duplex communication between two processes using forks, pipes, and exec system calls. The program reads from the original file (filename.inpf), runs enoding processes on the data and passes it to the other process which decodes, runs an uppercase function, and encodes the data again before sending it through the second pipe to the other process . The child process then receives the data, decodes it, and then prints the uppercase version to an output file.

### **System Requirements**

* **Operating System**: Linux-based (e.g., Ubuntu, CentOS)
* **C Compiler**: GCC (GNU Compiler Collection)
* **Required Files**:
  + filename.inpf: Input text file to be processed.
  + Source code files:
    - Assignment3.c
    - encDec.h

### **Troubleshooting**

* method calls did not match their declarations
  + - * Fix: check the return types and change them so that they match
* too many method parameters passed to truncate
  + - * be sure that tr method only has its options and declare the original file as the input. tr will automatically call with the input
* end of pipe may not have been closed
  + - * close(pipe1/2[0-1]) after using intended end of the pipe
* file not closed after usage
  + - * close(*filename*) after file’s usage is over
* bad file descriptor
  + - * be sure that the file descriptor is open before using or while it is still in use
* Segmentation fault
  + - * Ensure that every descriptor is being closed correctly
* tr- bad file descriptor
  + - * ensure that the file used for truncate is in the right mode

# **Test Documentation**

The program will be tested for Inter-process Communication that is to test the full-duplex communication between the child and parent processes using pipes. The program is developed using GNU versions of the C compiler.

**Test cases:**

Verifying that the encoding and decoding function work correctly, all necessary data is sent to a file, and the uppercase function applies correctly

**Test Case-1:**

Input:

A black rectangle with white text

Description automatically generated

Output:

A computer screen with many white and blue text

Description automatically generated

Code:

**Assignment3.c**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <fcntl.h>

#include "encDec.h"

int pipe1[2];

int pipe2[2];

FILE\* originalFile;

FILE\* originalFile2;

FILE\* binaryFileIn;

FILE\* binaryFileOut;

FILE\* upperCaseFile;

FILE\* finishingFile;

FILE\* inputFile;

void producerConsumerChild();

void consumerProducerParent();

int main(){

    //initialize the children

    pid\_t child1;

    //create the pipes and check to see if they failed

    if (pipe(pipe1) < 0 || pipe(pipe2) < 0){

        fprintf(stderr, "Pipe Failed");

        return 1;

    }

    //create the first child

    child1 = fork();

    //check the first child process

    if (child1 < 0){

        fprintf(stderr, "Fork Failed");

        return 1;

    }

    else if(child1 > 0){

        consumerProducerParent();

        wait(NULL);

    }

    else{

        producerConsumerChild();

    }

    return 0;

}

//defintion for a void function

void consumerProducerParent(){

        printf("Consumer-Producer: Parent Process\n");

        //close the unused ends of the pipes

        close(pipe1[1]);

        close(pipe2[0]);

        //save to revert standard output later

        int saved\_stdout = dup(fileno(stdout));

        if (saved\_stdout == -1) {

            perror("dup");

            exit(EXIT\_FAILURE);

        }

        //save to revert standard input later

        int saved\_stdin = dup(fileno(stdin));

        if (saved\_stdin == -1) {

            perror("dup");

            exit(EXIT\_FAILURE);

        }

        //--------------------------------------------------------------------------------------------------

        //parent is receiving the data from the child

        char catchFrames[100][255];

        int i = 0;

        while(readPipe(pipe1[0], catchFrames[i], sizeof(catchFrames[i])) > 0)

        {

            catchFrames[i][254] = '\0';

            i++;

        }

        close(pipe1[0]);

        //now let's decode each frame using a function

        char\* binaryWithParityBits = unconvertTheFrames(catchFrames);

        //and remove the parity bits

        char\* binaryMessage = removeParityBits(binaryWithParityBits);

        //and restore the old message back to its normal self

        char\* ogmessage = decodeMessage(binaryMessage);

        printf("Our original message is: %s\n", ogmessage);

        //---------------------------------------------------------------------------------------------

        //  OUTPUT FILE

        //put this message in an output file

        originalFile = fopen("output.txt", "r");

        if(originalFile == NULL){

            fprintf(stderr, "Error opening file");

            exit(EXIT\_FAILURE);

        }

        int write1 = fwrite(ogmessage, sizeof(char), strlen(ogmessage), originalFile);

        if(write1 < 0)

        {

            fprintf(stderr, "Error writing to file");

            exit(EXIT\_FAILURE);

        }

        //set the standard input file to the original file

        if (dup2(fileno(originalFile), STDIN\_FILENO) == -1) {

            perror("dup2");

            exit(EXIT\_FAILURE);

        }

        fflush(stdin);

        fclose(originalFile);

        //-----------------------------------------------------------------------------------

        // UPPERCASE FILE

        upperCaseFile = fopen("upperCase.txt", "a");

        if (upperCaseFile == NULL) {

            fprintf(stderr, "Error opening upperCase.txt");

            exit(EXIT\_FAILURE);

        }

        // Redirect stdout to the upperCase file

        if (dup2(fileno(upperCaseFile), STDOUT\_FILENO) == -1) {

            perror("dup2");

            exit(EXIT\_FAILURE);

        }

        toUpperCase("output.txt");

        fflush(upperCaseFile);

        // Close upperCaseFile and restore stdout

        fclose(upperCaseFile);

        // Restore stdout

        if (dup2(saved\_stdout, STDOUT\_FILENO) == -1) {

            perror("dup2");

            exit(EXIT\_FAILURE);

        }

        close(saved\_stdout);  // Close saved\_stdout after restoring

        //restore the original stdin (back to the console)

        if (dup2(saved\_stdin, STDIN\_FILENO) == -1) {

            perror("dup2");

            exit(EXIT\_FAILURE);

        }

        close(saved\_stdin);

        //we are now opening the input file

        upperCaseFile = fopen("upperCase.txt", "r");

        //check if the file opened successfully

        if(upperCaseFile == NULL){

            fprintf(stderr, "Error opening file");

            exit(EXIT\_FAILURE);

        }

        // Move to the end of the file to get the size

        fseek(upperCaseFile, 0, SEEK\_END);

        //get the size of the file

        long fileSize2 = ftell(upperCaseFile);

        fseek(upperCaseFile, 0, SEEK\_SET); // Reset to the beginning of the file

        printf("The file size is: %ld", fileSize2);

        char\* buffer2 = malloc(fileSize2 + 1);  // Allocate buffer dynamically to match file size

        if (buffer2 == NULL) {

            fprintf(stderr, "Memory allocation failed\n");

            exit(EXIT\_FAILURE);

        }

        fread(buffer2, sizeof(char), fileSize2, upperCaseFile);

        buffer2[fileSize2] = '\0';  // Null-terminate the string

        fclose(upperCaseFile);

        printf("Buffer2: %s\n", buffer2);

        //use a function to covert the buffer string to an array of bits

        char\* bitArray2 = encodeMessage(buffer2);

        //write bitArray to a file

        binaryFileOut = fopen("filename.chck", "w");

        if(binaryFileOut == NULL){

            fprintf(stderr, "Error opening file");

            exit(EXIT\_FAILURE);

        }

        int write2 = fwrite(bitArray2, sizeof(char), strlen(bitArray2), binaryFileOut);

        if(write2 < 0)

        {

            fprintf(stderr, "Error writing to file");

            exit(EXIT\_FAILURE);

        }

        fclose(binaryFileOut);

        //add the parity bits to the file

        char\* bitArrayWithParity2 = addParityBits(bitArray2);

        //split the modified bitArray into frames

        char\*\* frameArray2 = convertToFrames(bitArrayWithParity2);

        for(int i = 0; i < sizeof(frameArray2); i++)

        {

            printf("Frame %d: %s\n", i+1, frameArray2[i]);

            writePipe(pipe2[1], frameArray2[i], strlen(frameArray2[i]));

        }

        close(pipe2[1]);

}

void producerConsumerChild(){

        printf("Consumer-Producer: Child Process\n");

        //close the unused ends of the pipes

        close(pipe1[0]);

        close(pipe2[1]);

        //---------------------------------------------------------------------------

        //THE INPUT FILE

        //we are now opening the input file

        inputFile = fopen("filename.inpf", "r");

        //check if the file opened successfully

        if(inputFile == NULL){

            fprintf(stderr, "Error opening file");

            exit(EXIT\_FAILURE);

        }

        // Move to the end of the file to get the size

        fseek(inputFile, 0, SEEK\_END);

        long fileSize = ftell(inputFile);

        fseek(inputFile, 0, SEEK\_SET); // Reset to the beginning of the file

        printf("The file size is %ld", fileSize);

        char\* buffer = malloc(fileSize + 1);  // Allocate buffer dynamically to match file size

        if (buffer == NULL) {

            fprintf(stderr, "Memory allocation failed\n");

            exit(EXIT\_FAILURE);

        }

        fread(buffer, sizeof(char), fileSize, inputFile);

        buffer[fileSize] = '\0';  // Null-terminate the string

        //use a function to covert the buffer string to an array of bits

        char\* bitArray = encodeMessage(buffer);

        //write bitArray to a file

        binaryFileIn = fopen("filename.binf", "w");

        if(binaryFileIn == NULL){

            fprintf(stderr, "Error opening file");

            exit(EXIT\_FAILURE);

        }

        int write3 = fwrite(bitArray, sizeof(char), strlen(bitArray), binaryFileIn);

        if(write3 < 0)

        {

            fprintf(stderr, "Error writing to file");

            exit(EXIT\_FAILURE);

        }

        fclose(binaryFileIn);

        //add the parity bits to the file

        char\* bitArrayWithParity = addParityBits(bitArray);

        //split the modified bitArray into frames

        char\*\* frameArray = convertToFrames(bitArrayWithParity);

        for(int i = 0; i < sizeof(frameArray); i++)

        {

            printf("Frame %d: %s\n", i+1, frameArray[i]);

            writePipe(pipe1[1], frameArray[i], strlen(frameArray[i]));

        }

        fclose(inputFile);

        close(pipe1[1]);

        //-----------------------------------------------------------------------------------------------

        //parent is receiving the data from the child

        char catchFrames2[100][255];

        printf("We have reached");

        int i = 0;

        while(readPipe(pipe2[0], catchFrames2[i], sizeof(catchFrames2[i])) > 0)

        {

            catchFrames2[i][254] = '\0';

            i++;

        }

        //close the read end of the pipe

        close(pipe2[0]);

        //now let's decode each frame using a function

        char\* binaryWithParityBits2 = unconvertTheFrames(catchFrames2);

        //and remove the parity bits

        char\* binaryMessage2 = removeParityBits(binaryWithParityBits2);

        //and restore the old message back to its normal self

        char\* ogmessage2 = decodeMessage(binaryMessage2);

        printf("Our original message sent back over is: %s\n", ogmessage2);

        //put this message in an output file

        originalFile2 = fopen("output2.txt", "w");

        if(originalFile2 == NULL){

            fprintf(stderr, "Error opening file");

            exit(EXIT\_FAILURE);

        }

        int write4 = fwrite(ogmessage2, sizeof(char), strlen(ogmessage2), originalFile2);

        if(write4 < 0)

        {

            fprintf(stderr, "Error writing to file");

            exit(EXIT\_FAILURE);

        }

        fclose(originalFile2);

        //write bitArray to a file

        finishingFile = fopen("filename.done", "w");

        if(finishingFile == NULL){

            fprintf(stderr, "Error opening file");

            exit(EXIT\_FAILURE);

        }

        fwrite(ogmessage2, sizeof(char), strlen(ogmessage2), finishingFile);

        fclose(finishingFile);

}

**encDec.h**

#ifndef ENCDEC\_H

#define ENCDEC\_H

#include <stddef.h>

#include <string.h>

#include <stdio.h>

#define CHAR\_BITS 8

//definition for the readPipe function

ssize\_t readPipe(int fd, void\* buf, size\_t count){

    return read(fd, buf, count);

}

//definition for the writePipe function

ssize\_t writePipe(int fd, const void\* buf, size\_t count){

    return write(fd, buf, count);

}

//converts the input string to bits

char\* encodeMessage(const char\* inputbuffer)

{

    printf("Our input: %s\n", inputbuffer);

    //the size of the original message

    size\_t length = strlen(inputbuffer);

    size\_t bitArrayLength = length \* CHAR\_BITS;

    char\* bitArray = (char\*)malloc((bitArrayLength + 1) \* sizeof(char));

    if (!bitArray) {

        perror("malloc failed");

        return NULL;

    }

    int index = 0;

    for (size\_t i = 0; i < length; i++) {

        for (int j = CHAR\_BITS - 1; j >= 0; j--) {

            bitArray[index++] = (inputbuffer[i] >> j) & 1 ? '1' : '0';

        }

    }

    bitArray[bitArrayLength + 1] = '\0';

    printf("Encode Message: %s\n",bitArray);

    return bitArray;

}

char\* decodeMessage(char\* encodedMessage) {

    int length = strlen(encodedMessage);

    // Make sure the binary string is a multiple of 8

    if (length % 8 != 0) {

        fprintf(stderr, "Invalid binary string length. Must be a multiple of 8.\n");

        return NULL;

    }

    // Allocate memory for the ASCII string (length / 8 characters + null terminator)

    int asciiLength = length / 8;

    char\* asciiString = malloc(asciiLength + 1); // +1 for the null terminator

    if (asciiString == NULL) {

        fprintf(stderr, "Memory allocation failed.\n");

        return NULL;

    }

    // Iterate over the binary string in chunks of 8

    for (int i = 0; i < length; i += 8) {

        char byte[9] = {0}; // Temporary array to hold each byte (8 bits + null terminator)

        // Copy 8 bits into the byte array

        strncpy(byte, encodedMessage + i, 8);

        // Convert the binary string to an integer

        int asciiValue = (int)strtol(byte, NULL, 2);

        // Store the corresponding ASCII character

        asciiString[i / 8] = (char)asciiValue;

    }

    // Null-terminate the ASCII string

    asciiString[asciiLength] = '\0';

    return asciiString;

}

//modify the binary string to have parity bits

char\* addParityBits(char\* bitArray)

{

    //this is where we will add the parity bit to each character

    for(int i = 0; i < strlen(bitArray); i += 8)

    {

        int count1 = 0;

        for (int j = i; j < (i + 8); j++)

        {

            if(bitArray[j] == '1')

            {

                count1++;

            }

        }

        if(count1 % 2 == 0)

        {

            bitArray[i] = '1';

        }

    }

    return bitArray;

}

char\* removeParityBits(char\* bitArray){

    for(int i = 0; i < strlen(bitArray); i += 8)

    {

        bitArray[i] = '0';

    }

    return bitArray;

}

//convert the input bits to frames

char\*\* convertToFrames(char\* bitArray)

{

    //do a cealing function for the number of frames

    int numFrames = strlen(bitArray) / 64;

    if(strlen(bitArray) % 64 != 0)

    {

        numFrames += 1;

    }

    //allocate memory for the frameArray and copy the bits to it

    char\*\* frameArray = (char\*\*)malloc(numFrames \* sizeof(char\*));

    for(int i = 0; i < numFrames; i++){

        frameArray[i] = (char\*)malloc(89);

        if(!frameArray[i]){

            perror("malloc failed");

            return NULL;

        }

        //set the first 16 bits of the frame to be "2222"

        strncpy(frameArray[i], "0001011000010110", 17);

        frameArray[i][16] = '\0';

        if((strlen(bitArray) % 64 != 0) && ((i == (numFrames - 1))))

        {

            //We are now handling the last frame

            int remainder = strlen(bitArray) % 64;

            // Convert the remainder to binary

            char binaryString[8];

            binaryString[0] = '\0'; // Initialize the binary string

            int index = 0;

            // Generate binary representation

            for (int i = 7; i >= 0; i--) {

                if (remainder & (1 << i)) {

                    binaryString[index++] = '1';

                } else { // Start adding '0's after the first '1'

                    binaryString[index++] = '0';

                }

            }

            // If no bits were added, the number is 0

            if (index == 0) {

                binaryString[index++] = '0';

            }

            binaryString[index] = '\0';

            strncat(frameArray[i], binaryString, 9);

            frameArray[i][24] = '\0';

        }

        else{

            strncat(frameArray[i], "01000000", 9);

            frameArray[i][24] = '\0';

        }

        strncat(frameArray[i], bitArray, 64);

        frameArray[i][88] = '\0';

    }

    //count the number of ones in frameArray

    return frameArray;

}

//definition of decodeFrames which decodes each frame received

char\* unconvertTheFrames(char frameArray[100][255])

{

    char\* originalMessage = malloc(1024);

    if (originalMessage == NULL) {

        fprintf(stderr, "Memory allocation failed\n");

        exit(EXIT\_FAILURE);

    }

    originalMessage[0] = '\0';

    int i = 0;

    while(frameArray[i][0] != '\0')

    {

        //fetch the message length section from our frame

        char bitCount[8] = {0};

        for(int j = 16; j < 24; j++) {

            if(frameArray[i][j] == '1') {

                bitCount[j - 16] = '1';

            } else {

                bitCount[j - 16] = '0';

            }

        }

        //convert this binary string into a decimal value

        int decimalValue = 0;

        for (int j = 0; j < strlen(bitCount); j++) {

            // Shift the current decimal value left by 1 (multiply by 2)

            decimalValue <<= 1;

            // Add 1 if the current bit is '1'

            if (bitCount[j] == '1') {

                decimalValue |= 1;

            }

        }

        char messagepiece[1024] = ""; // Initialize messagepiece as an empty string

        int messageIndex = 0; // Index to track position in messagepiece

        for (int j = 0; j < decimalValue; j++) {

            if (frameArray[i][j + 24] == '1') {

                messagepiece[messageIndex++] = '1';

            } else {

                messagepiece[messageIndex++] = '0';

            }

        }

        // Null-terminate messagepiece

        messagepiece[messageIndex] = '\0';

        printf("Our original binary string with parity bits was: %s\n", messagepiece);

        // Check if there's enough space in originalMessage before concatenation

        if (strlen(originalMessage) + strlen(messagepiece) < 1024) {

            strcat(originalMessage, messagepiece); // Concatenate safely

        } else {

            fprintf(stderr, "Original message overflow risk\n");

            free(originalMessage);

            exit(EXIT\_FAILURE);

        }

        i++;

    }

    // Print the final decoded message

    printf("Final original message: %s\n", originalMessage);

    return originalMessage;

}

//definition for the toLower function that returns a string

int toUpperCase(char\* input){

    //read and convert the file contents to lowercase

    pid\_t child;

    child = fork();

    if(child < 0){

        fprintf(stderr, "Fork Failed");

        return -1;

    }

    else if(child > 0){

        wait(NULL);

        return -1;

    }

    else{

        //convert the characters to lower case in the child process

        return execl("/usr/bin/tr", "tr", "a-z", "A-Z", (char\*)NULL);

    }

}

#endif