University at Albany

Department of Computer Science

CSI 500- OPERATING SYSTEMS

PROJECT DOCUMENTATION

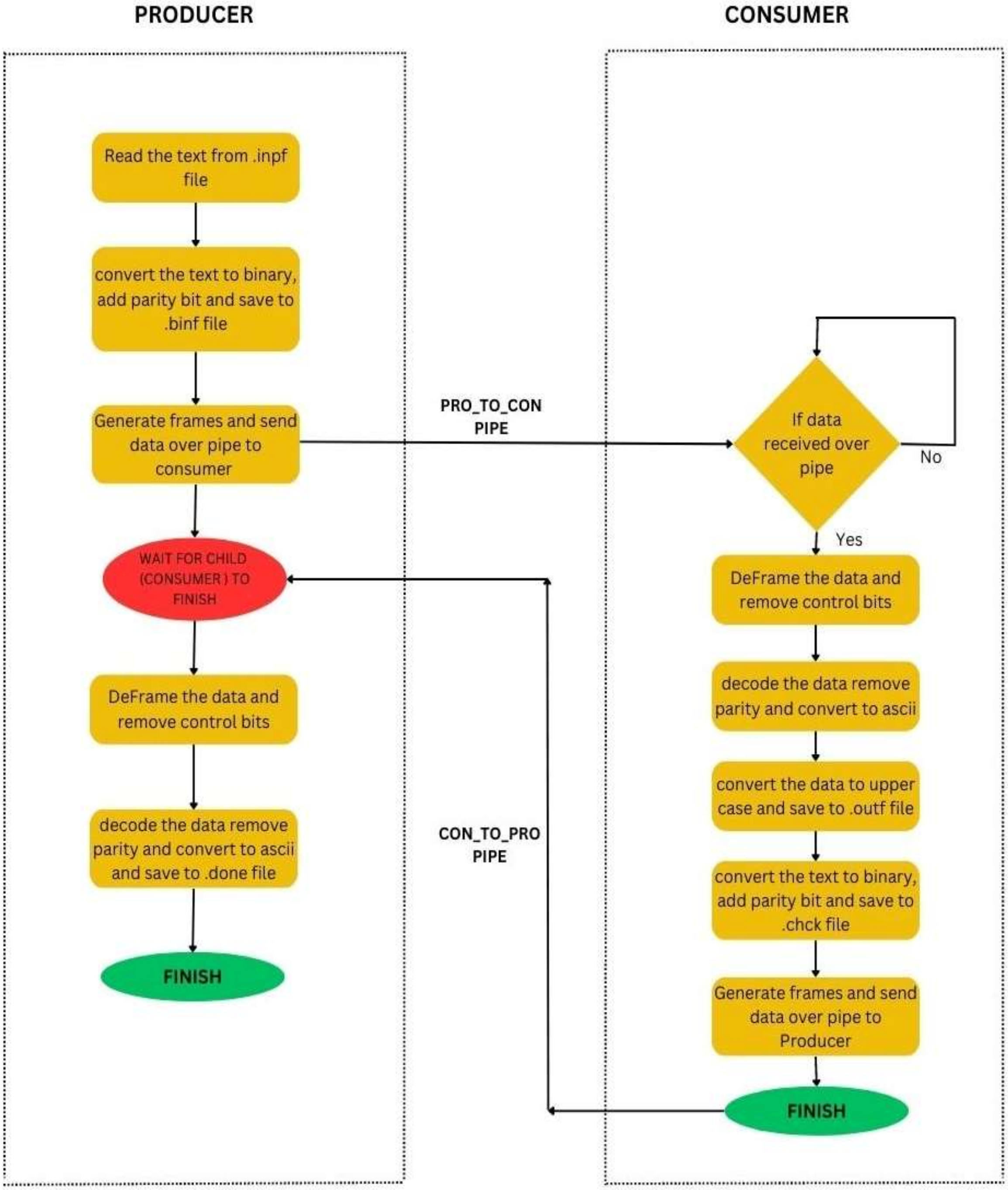
--

Submitted by: CHARAN KASALA

b) Table of Contents

c) SYSTEM DOCUMENTATION:

(i) A high-level data flow diagram for the system


PRODUCER
CONSUMER
Read the text from .inpf file
convert the text to binary, add parity bit and save to .binf file
Generate frames and send data over pipe to consumer
PRO_TO_CON PIPE
If data received over pipe
No
Yes
WAIT FOR CHILD (CONSUMER) TO FINISH
DeFrame the data and remove control bits
decode the data remove parity and convert to ascii
convert the data to upper case and save to .outf file
convert the text to binary, add parity bit and save to .chck file
Generate frames and send data over pipe to Producer
FINISH
CON_TO_PRO PIPE
DeFrame the data and remove control bits
decode the data remove parity and convert to ascii and save to .done file
FINISH

(ii) List of Routines or functions used and their Description:

1) char *char_to_binary7(char c)

This function is used to a convert single character to 7-bit binary string.

2) char *char_to_binary8(char c)

This function is used to convert a single character to 8-bit binary string.

3) char* generate_prefix()

This function to used to generate the header for frame based on SYN CHAR and LENGTH OF DATA BLOCK based on the data given in encDec.h header file and writes the prefix in to Header-Frame.binf file.

4) char* add_parity_bit(char binary[BUFFER_SIZE])

This function gets the 7-bit binary array as input parameter and checks the parity. If number of 1's in the binary array is even, then 1 is added to the MSB bit of the array and returned. If it is odd, then 0 is added to the MSB bit of the array and returned .

5) int enc_data(char* input_filename, char* output_filename)

This module reads the data from the input file, converts each character to a 7-bit binary string, checks and adds parity bit to the MSB of the string and writes the data into the output file name.

6) void transfer_data(int fd, char* data, int len)

The function is used for sending framed data over the pipes from producer to consumer and consumer to producer

7) char* frame_creation(char *input, int remaining_len, char *prefix)

This function is used for adding header (control characters) to data characters based on the data in Header-Frame.binf and creates frames for transfer between producer and consumer.

8) int generate_frames(char* input_filename,int pipefd[2])

This module handles the logic for dividing the encoded data into chunks and uses frame_creation () method to create frames and forwards the frames generated to transfer_data function to transfer the data using pipe.

9) void receive_data(int fd, char* data, int len)

This function is used for receiving framed data over the pipes from producer to consumer and consumer to producer.

10) char* frame_removal(char *input, char *prefix)

This function is used for removing header (control characters) from the data received from the pipe and sends the data to deFrame() module.

11) int deFrame(int pipefd[2], char* output_filename)

This module receives the framed data from the pipe between consumer and producer and then removing the frames using frame_removal function and writes the decoded data to temporary binf file.

12) char* chk_remove_parity(char* binary_string, int n)

This function is used to check for the parity bits in the given string and remove the parity bits. If the parity is not matching, then it displays error message for Invalid parity.

13) char* binary_to_ascii(char* binary_string);

This function is used for converting the binary string into ASCII character.

14) int dec_data(char* input_filename, char* output_filename);

This module reads the data from the input file, checks for parity and removes parity using chk_remove_parity() method and converts each 7-bit string into ASCII character and writes the data into the output file name.

15) int convert_toupper(char* filename)

This function is used for reading all the contents of a file and converts each character into upper case and then write it back to the file.

16) encDec.h

This header file contains the prototypes of all the code required to support the producer consumer application.

17) execl() :

execl() is one of the functions included in the exec family functions. After executing execl(), replaces the current process image with a new process image specified by path. In the ProducerConsumer application developed by me execl() function was used to execute the different layers of the application as per the requirements in the Project Requirement document.

18) pipe():

This function creates a pipe and puts the file descriptors for the reading and writing end of the pipe into pfd[0] & pfd[1] respectively.

19) fork ():

A new process known as a "child process" is created by invoking the fork system function and the child runs concurrently with the process that invoked fork () (i.e., parent process). Both processes will carry out the next instruction after the fork () system call once a new child process has been started. The Child process utilizes the same CPU registers, program counter, and open files that the parent process utilizes.

20) Wait ():

Until one of its child processes ends or a signal is received, a call to wait () pauses the caller process (i.e., Parent Process). The Parent process continues to run after the wait system call command after the child process exits.

21)close ():

The close () function deallocates file descriptor indicated by fields. To deallocate means to make the file descriptor available for return by subsequent calls to open () or other functions that allocate file descriptors.

22) exit ():

The calling process is instantly terminated using the C library function void exit(int status). The process's open file descriptors are closed, its children if any are inherited by process 1, init, and the process parent receives a SIGCHLD signal.

23) strcpy():

This function copies the string pointed by source (including the null character) to the destination.

24) strcat()

The strcat() function concatenates the destination string and the source string, and the result is stored in the destination string.

In addition to above mentioned functions and routines several other c language function were used for the implementation of the project

(iii) Implementation details:

I have created a the following c program files and header files for implementation of project 2.

1) encDec.h
2) generatePrefix.c
3) convertToUpper.c
4) encodeService.c
5) decodeService.c
6) ProducerConsumer.c

encDec.h is a header file that contains all the function prototypes which were used in the implementation of the project 2.

I have included the following header files in my c programs which are pre-defined c libraries.

```
<stdio.h>
<string.h>
<stdlib.h>
<unistd.h>
<errno.h>
#include <ctype.h>
#include <sys/wait.h>
#include "encDec.h"
```

samplefile.inpf file is given as input argument while executing the ProducerConsumer application.

The project starts by executing the main function of ProducerConsumer.c program. Based on the SYN_CHAR and BLOCK_LEN specified by the user in "encDec.h" header file, the control characters that are to be added in the headers of each frame are generated by executing the generatePrefix() executable file using the execl() system call and then stored in Header-Frame.binf.

Two pipes pro_to_con and con_to_pro are then created for handling transfer of frames between producer and consumer. Then the input data contained in the samplefile.inpf is taken as input by the Producer and each character is converted into binary code and then based on the scheme of odd parity, parity bit is appended to the MSB of each individual binary string. The data is then stored on the samplefile.binf file. The above operations are performed by executing the encodeService() executable service using the execl() system call.

The control characters generated (prefix) are then added to each chunk based on the block length specified. The data is then passed on to the Consumer by writing the frames to the write end of the pro_to_con pipe. The write end of the pro_to_con pipe is then closed.

The framed data is then received by the Consumer by opening the read end of the pro_to_con file and then all the header data of the frames received are removed by reading the control characters from Header-Frame.binf and the data is written on to the temporary file o_after_deframe.binf.

The data is then read from the file by the Consumer and then parity is checked and removed. If there is any difference in parity bits, then error message is displayed. The data is then decoded into ascii characters. The above operations are performed by executing the decodeService() executable service using the execl() system call.

The decoded data is then converted into upper case characters by executing the convertToUpper() executable service using the execl() system call and stored into samplefile.outf.

Then the data contained in the samplefile.outf is taken as input by the Consumer and each character is converted into binary code and then based on the scheme of odd parity, parity bit is appended to the MSB of each individual binary string. The data is then stored on the samplefile.chck file. The above operations are performed by executing the encodeService() executable service using the execl() system call.

The control characters generated (prefix) are then added to each chunk based on the block length specified. The data is then passed on to the Producer by writing the frames to the write end of the con_to_pro pipe. The write end of the con_to_pro pipe is then closed.

The framed data is then received by the Producer by opening the read end of the con_to_pro file and then all the header data of the frames received are removed by reading the control characters from Header-Frame.binf and the data is written on to the temporary file o_after_deframe2.binf.

The data is then read from the file by the Producer and then parity is checked and removed. If there is any difference in parity bits, then error message is displayed. The data is then decoded into ascii characters. The above operations are performed by executing the decodeService() executable service using the execl() system call and stored into samplefile.done.

With this the implementation of ProducerConsumer application as specified in the Project Requirement document is done successfully and the final desired output is present in samplefile.done . This can be accessed by the user to check the correctness of the data.

d) TEST DOCUMENTATION:

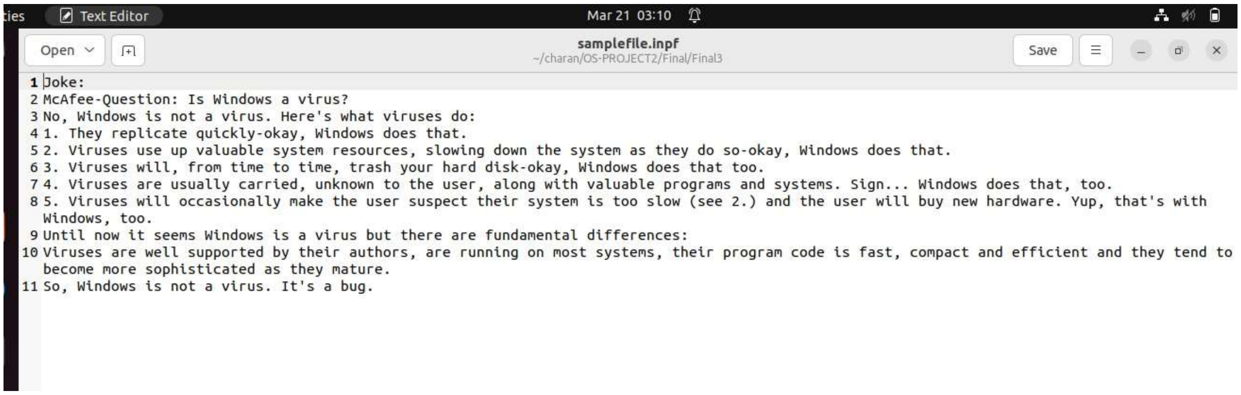
(i) How the program was tested:

As the program is tested in a GCC Compiler, the executable file will be created which shall be executed to get the output.

The input file samplefile.inpf is required for testing the program.

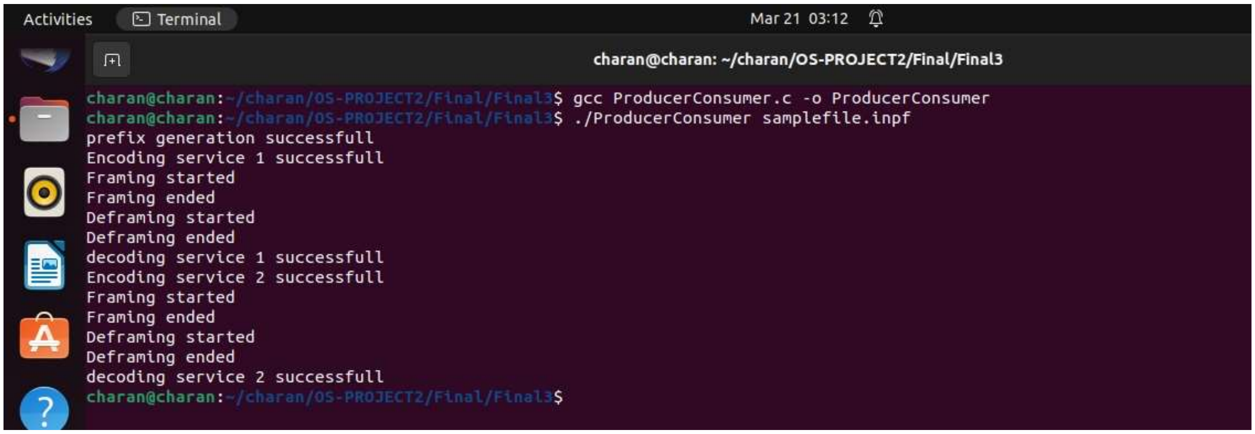
I have tested the program with the sample data inside samplefile.inpf as given in the project requirement document (i.e., windows joke)

samplefile.inpf



```
Joke:
McAfee-Question: Is Windows a virus?
No, Windows is not a virus. Here's what viruses do:
1. They replicate quickly-okay, Windows does that.
2. Viruses use up valuable system resources, slowing down the system as they do so-okay, Windows does that.
3. Viruses will, from time to time, trash your hard disk-okay, Windows does that too.
4. Viruses are usually carried, unknown to the user, along with valuable programs and systems. Sign... Windows does that, too.
5. Viruses will occasionally make the user suspect their system is too slow (see 2.) and the user will buy new hardware. Yup, that's with Windows, too.
Until now it seems Windows is a virus but there are fundamental differences:
Viruses are well supported by their authors, are running on most systems, their program code is fast, compact and efficient and they tend to become more sophisticated as they mature.
So, Windows is not a virus. It's a bug.
```

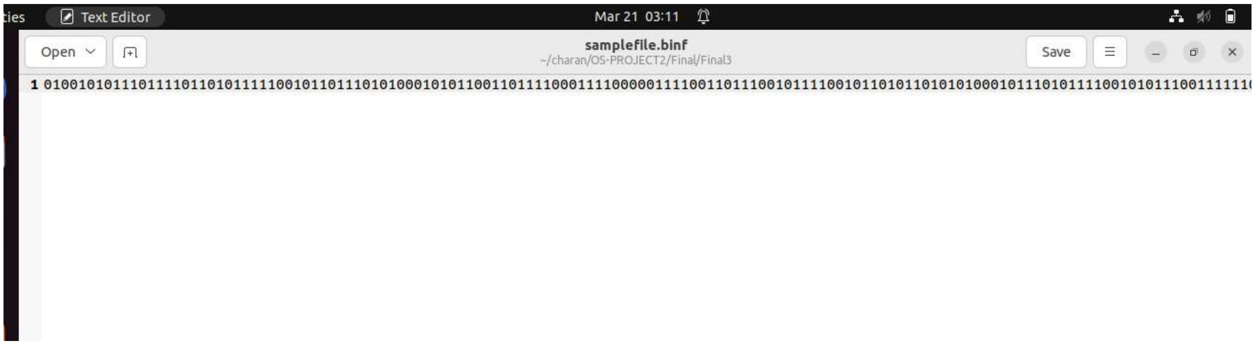
When the producer consumer application was executed by giving samplefile.inpf as input file, the application was executed successfully as shown below.



```
charan@charan:~/charan/OS-PROJECT2/Final/Final3$ gcc ProducerConsumer.c -o ProducerConsumer
charan@charan:~/charan/OS-PROJECT2/Final/Final3$ ./ProducerConsumer samplefile.inpf
prefix generation successful
Encoding service 1 successfull
Framing started
Framing ended
Deframing started
Deframing ended
decoding service 1 successfull
Encoding service 2 successfull
Framing started
Framing ended
Deframing started
Deframing ended
decoding service 2 successfull
charan@charan:~/charan/OS-PROJECT2/Final/Final3$
```

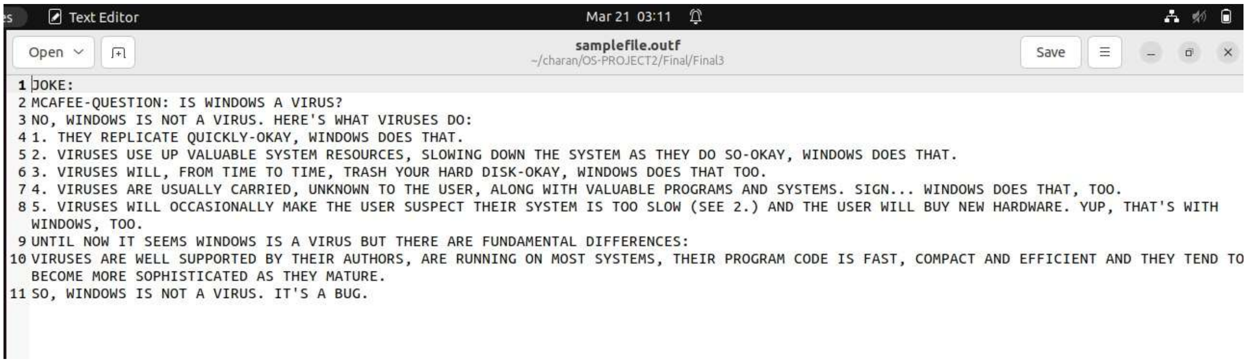
After execution the following files are generated as per the instructions given in the project requirements document

samplefile.binf



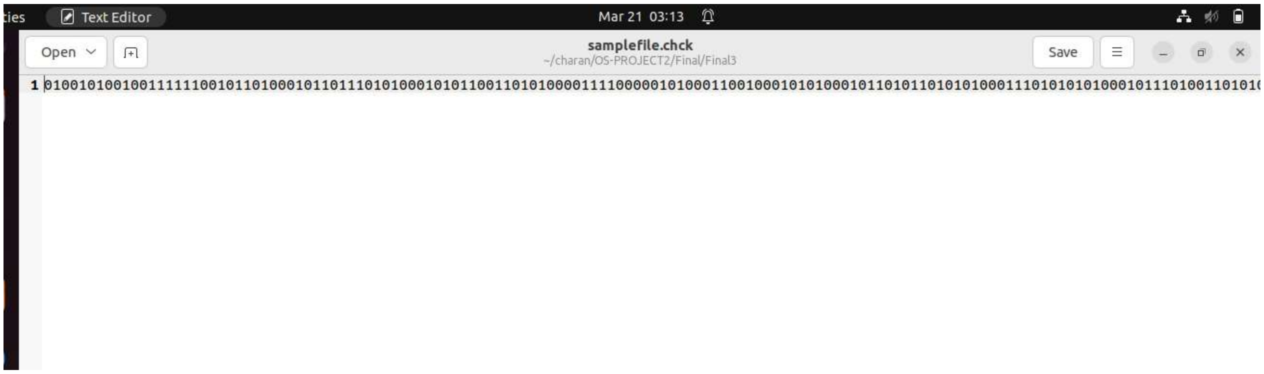
```
0100101011101111011010111110010110111010100010101100110111100011110000011110011011100101111001011010110101010001011101011110010101110011111
```

samplefile.outf



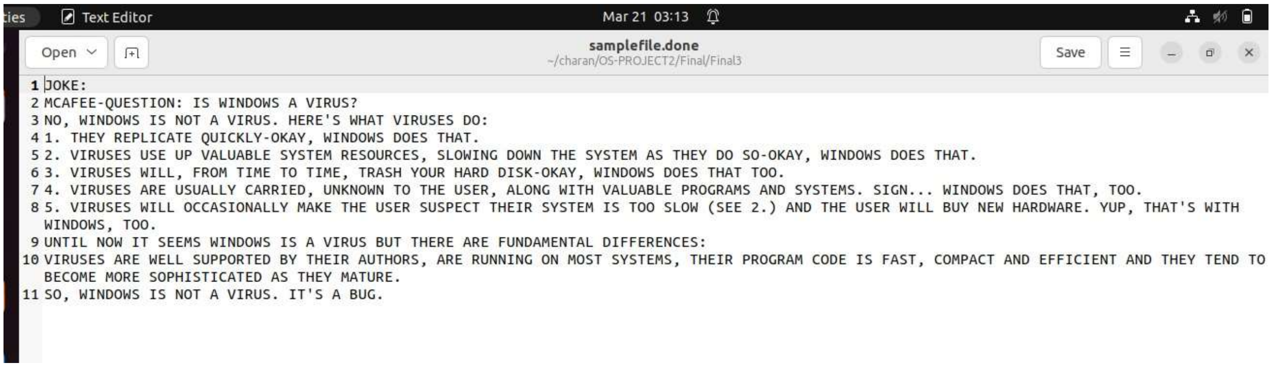
```
JOKE:
MCAFEE-QUESTION: IS WINDOWS A VIRUS?
NO, WINDOWS IS NOT A VIRUS. HERE'S WHAT VIRUSES DO:
1. THEY REPLICATE QUICKLY-OKAY, WINDOWS DOES THAT.
2. VIRUSES USE UP VALUABLE SYSTEM RESOURCES, SLOWING DOWN THE SYSTEM AS THEY DO SO-OKAY, WINDOWS DOES THAT.
3. VIRUSES WILL, FROM TIME TO TIME, TRASH YOUR HARD DISK-OKAY, WINDOWS DOES THAT TOO.
4. VIRUSES ARE USUALLY CARRIED, UNKNOWN TO THE USER, ALONG WITH VALUABLE PROGRAMS AND SYSTEMS. SIGN... WINDOWS DOES THAT, TOO.
5. VIRUSES WILL OCCASIONALLY MAKE THE USER SUSPECT THEIR SYSTEM IS TOO SLOW (SEE 2.) AND THE USER WILL BUY NEW HARDWARE. YUP, THAT'S WITH WINDOWS, TOO.
UNTIL NOW IT SEEMS WINDOWS IS A VIRUS BUT THERE ARE FUNDAMENTAL DIFFERENCES:
VIRUSES ARE WELL SUPPORTED BY THEIR AUTHORS, ARE RUNNING ON MOST SYSTEMS, THEIR PROGRAM CODE IS FAST, COMPACT AND EFFICIENT AND THEY TEND TO BECOME MORE SOPHISTICATED AS THEY MATURE.
SO, WINDOWS IS NOT A VIRUS. IT'S A BUG.
```

samplefile.chck



```
010010100100111111001011010001011011101010001010110011010100001111000001010001100100010101000101101011010101000111010101010001011101001101010
```

samplefile.done



```
JOKE:
MCAFEE-QUESTION: IS WINDOWS A VIRUS?
NO, WINDOWS IS NOT A VIRUS. HERE'S WHAT VIRUSES DO:
1. THEY REPLICATE QUICKLY-OKAY, WINDOWS DOES THAT.
2. VIRUSES USE UP VALUABLE SYSTEM RESOURCES, SLOWING DOWN THE SYSTEM AS THEY DO SO-OKAY, WINDOWS DOES THAT.
3. VIRUSES WILL, FROM TIME TO TIME, TRASH YOUR HARD DISK-OKAY, WINDOWS DOES THAT TOO.
4. VIRUSES ARE USUALLY CARRIED, UNKNOWN TO THE USER, ALONG WITH VALUABLE PROGRAMS AND SYSTEMS. SIGN... WINDOWS DOES THAT, TOO.
5. VIRUSES WILL OCCASIONALLY MAKE THE USER SUSPECT THEIR SYSTEM IS TOO SLOW (SEE 2.) AND THE USER WILL BUY NEW HARDWARE. YUP, THAT'S WITH
WINDOWS, TOO.
UNTIL NOW IT SEEMS WINDOWS IS A VIRUS BUT THERE ARE FUNDAMENTAL DIFFERENCES:
VIRUSES ARE WELL SUPPORTED BY THEIR AUTHORS, ARE RUNNING ON MOST SYSTEMS, THEIR PROGRAM CODE IS FAST, COMPACT AND EFFICIENT AND THEY TEND TO
BECOME MORE SOPHISTICATED AS THEY MATURE.
SO, WINDOWS IS NOT A VIRUS. IT'S A BUG.
```

(ii)Test cases are along with the input and output files are attached in zip file

The ProducerConsumer application was also tested with 2 other input data files and the overall test cases are attached in below zip folder.

TestCases.zip

e) USER DOCUMENTATION:

(i) Running the program:

GNU versions of the C compiler was used to create the code. It's well-commented, layered, and modularized.

To execute the program please make sure that all the below mentioned files are in the same folder (or) directory.

1) encDec.h
2) generatePrefix.c
3) convertToUpper.c
4) encodeService.c
5) decodeService.c
6) ProducerConsumer.c
7) Samplefile.inpf

After making sure that the files are present in the same directory open the command prompt in the directory start compiling the source files as detailed below

Now the following steps are to be followed for compiling of the above c files.

```
$ gcc   generatePrefix.c -o generatePrefix
$ gcc   convertToUpper.c -o convertToUpper
$ gcc   encodeService.c -o encodeService
$ gcc   decodeService.c -o decodeService
$ gcc   ProducerConsumer.c -o ProducerConsumer
```

Now run the binary file by giving the samplefile.inpf as input argument to the file

```
$  ./ProducerConsumer samplefile.inpf
```

After executing the above commands, we can see that the following files are formed as per the instructions contained in the project requirement document

1) samplefile.binf
2) samplefile.outf
3) samplefile.chck
4) samplefile.done

The user can also change the data in samplefile.**inpf** and run the program again to generate the above files where samplefile.done contains the final output as specified in the project requirement document.

(ii) Parameters (if any):

samplefile.inpf file is to be passed as parameter to the ProducerConsumer application