# 

# **University at Albany**

# **Department of Computer Science**

# **CSI 500- OPERATING SYSTEMS**

**PROJECT DOCUMENTATION**

**-------------------------------------------------------------------------**

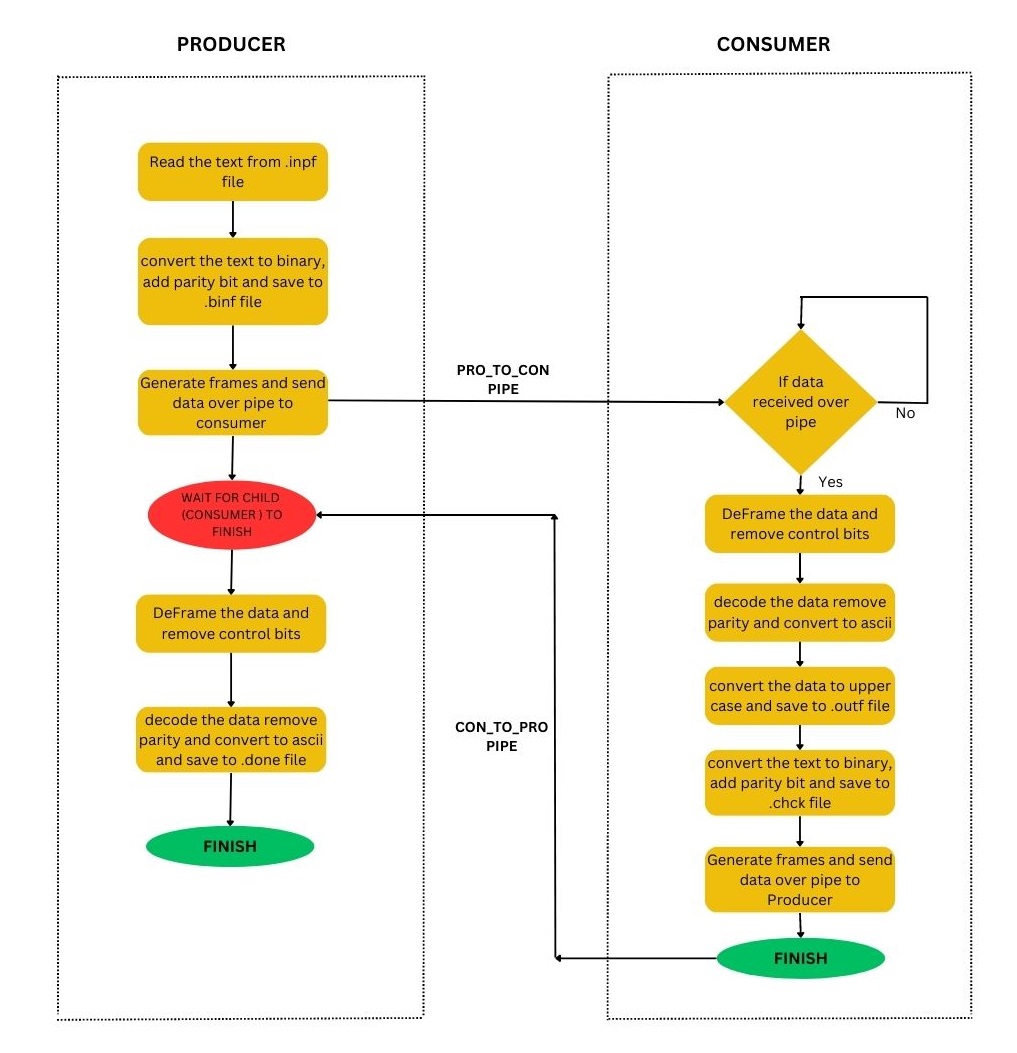
**Submitted by:** CHARAN KASALA

**b) Table of Contents**

|  |  |  |
| --- | --- | --- |
| **Sl No** | **Topic** | **Page No** |
| **c** | **SYSTEM DOCUMENTATION** | 3 |
| (i) A high-level data flow diagram for the system | 3 |
| (ii) A list or routines or functions used and their brief descriptions | 4 |
| (iii) Implementation details | 7 |
| **d** | **TEST DOCUMENTATION** | 10 |
| (i) How the program was tested | 10 |
| (ii) A list of your test sets including the results obtained by each set | 12 |
| **e** | **USER DOCUMENTATION** | 12 |
| (i) Running the program | 12 |
| (ii) Describe any parameters (if any) | 13 |

# **c) SYSTEM DOCUMENTATION:**

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



## **(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:**

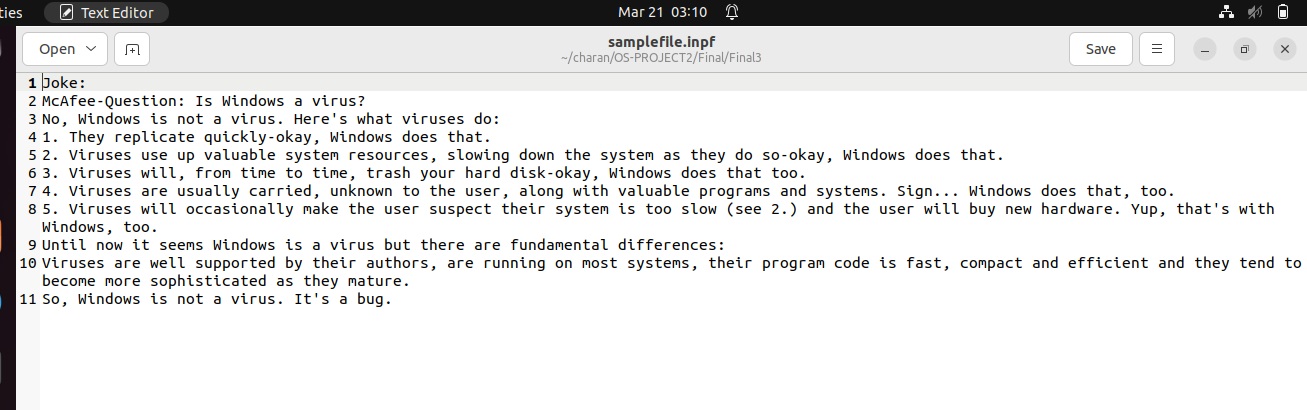
**(i) How the program was tested:**

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**



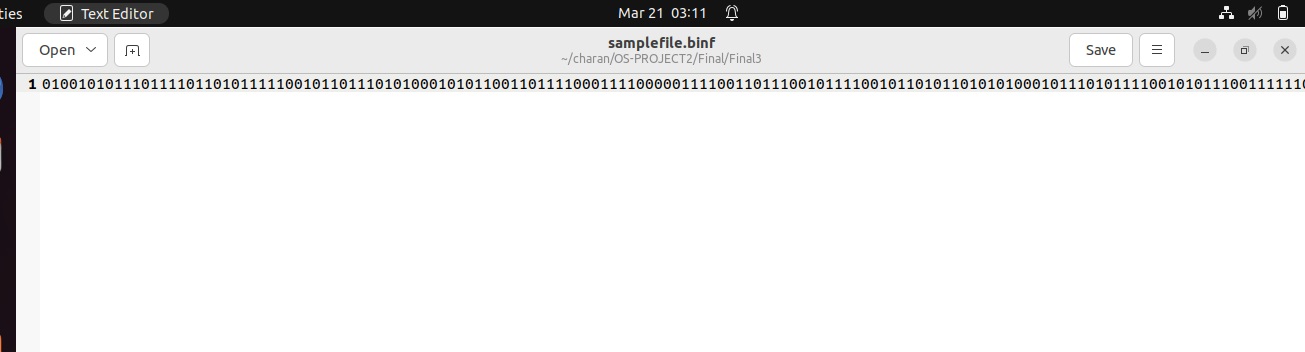
When the producer consumer application was executed by giving samplefile.inpf as input file, the application was executed successfully as shown below.

Text

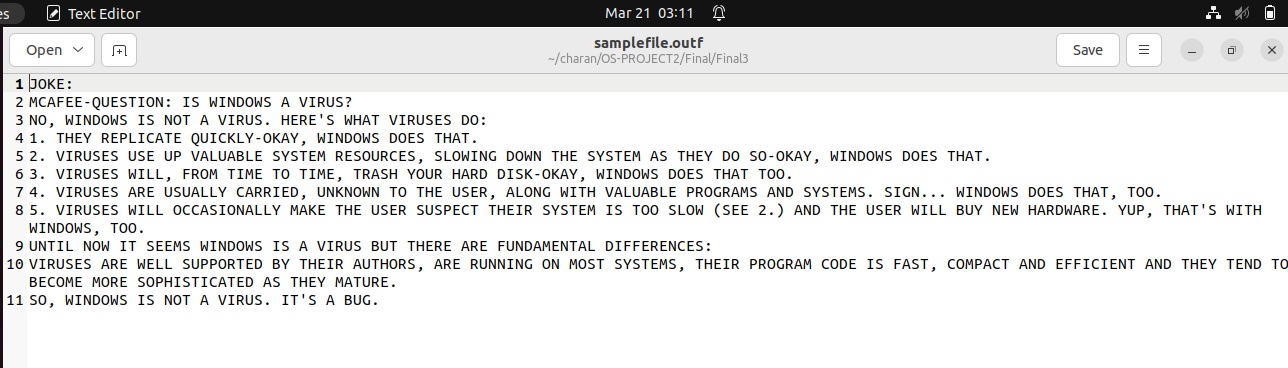
Description automatically generated

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

**samplefile.binf**



**samplefile.outf**

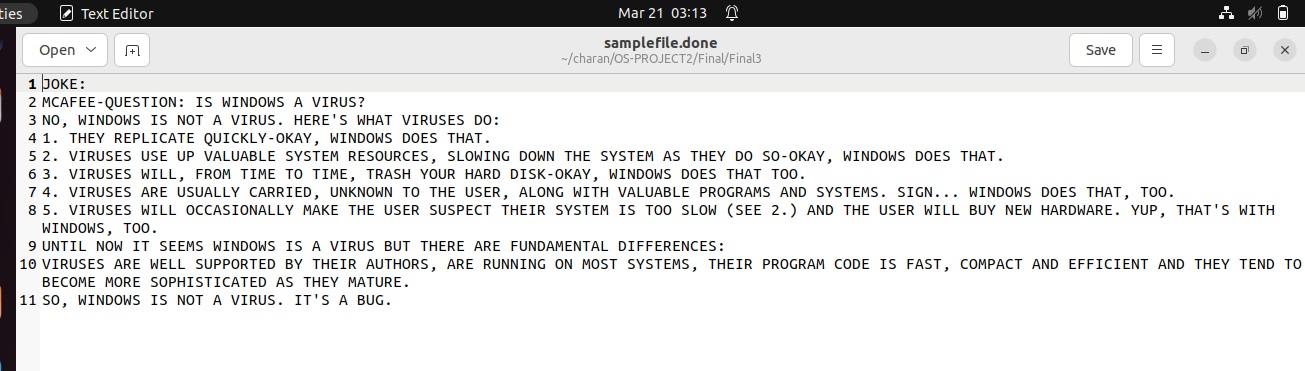
****

**samplefile.chck**

Graphical user interface, text

Description automatically generated

**samplefile.done**

****

**(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.



# **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