Operating Systems Project 2- *Draft Version*

University at Albany
Department of Computer Science
CSI 500

Project 2
Assigned: Tuesday, September 27th, 2022
Due: Thursday, October 20th by 11:59 PM. Submissions with 20% penalty will be Accepted by Tuesday, October 25th by 11:59 PM.

Purpose
To develop a synchronous inter-process communication through the use of both fork(), pipe(), and exec() system calls. For this exercise only **ordinary pipes** are to be used.

What to Do
You are to develop a producer/consumer application. Both the C programming language and any distribution of the Linux operating system are to be used. Your solution must be structured as a set of services. The core service is the inter-process communication which it is referred here as the producer/consumer service.

Details
A synchronous producer/consumer pair of processes is to be created to support the service of replacing all lowercase letters in a set of strings of characters with their uppercase correspondent. The service above described will require support of a set of other services to perform

1. the encoding and decoding of the data,
2. to change lowercase letters into uppercase,
3. to open and close files,
4. to include and verify parity bits,
5. to read from pipes, and
6. to write to pipes.

Both the standard ASCII character set and odd bit parity are to be used. All data communications between producer and consumer are to be done by reading and writing through two shared pipes. The data generated by the producer is encoded and shared with the consumer through one pipe. The consumer decodes the message, modifies it, encodes it, and shares it with the producer through the second pipe.

You are to create a library to support your producer/consumer process activities. The file "**encDec.h**" is to be created and it must contain the prototypes of all code required to support the producer/consumer activities as described by the services provided by the three layers discussed below. All access to the services provided by this library must be done only through the use of any of the members of the exec() family of system calls. The three distinct layers to be created are **physical**, **data link** and **application** and details regarding the services each one of them provided are as follows:

1. The **physical layer** must contain routines to handle tasks such as
    a. converting a character into a binary bit pattern,

   b. converting binary patterns into characters,
   c. including a parity bit, and
   d. checking and removing the parity bit.
2. The **data link layer** must contain routines for
   a. framing (putting two SYN characters, a LENGTH character, and data into a frame), and
   b. deframing (separating control characters from data characters).
3. The **application layer** must contain routines to handle tasks such as
   a. reading data to be transmitted from input data files in the producer, and
   b. writing received data into display (or output data file) in the consumer.

Your solution must also include a module to simulate transmission errors.

Information containing the original data is to be stored in a file with extension *inpf*. You are to use a set of temporary files to debug and document the results produced by the various activities performed during the processing of the required services. The encoded version of the input data, as well as the modified version of it are to be stored in files with the extension *binf*, and *outf* respectively. Files with the extensions *chck* and *done* will contain the encoded version of the file modified by the consumer and the decoded version of the data obtained by reading one of the pipes created respectively. Details regarding such files are provided by the table below.

File Characteristics			
Naming	**Contents**	**Created by**	**Accessed by**
filename.inpf	Any ASCII character	user	producer/consumer
filename.binf	Encoded (0/1) + parity + frame version of .inpf	producer	consumer
filename.outf	Uppercase version of .inpf	consumer	consumer
filename.chck	Encoded (0/1) + parity + frame version of .outf	consumer	consumer/producer
filename.done	Decoded + Deframed + no-parity version of .chck	producer	user

The Producer

Creates strings of bits (‘0’ and ‘1’ characters) from the input files. All valid input files contain data made up of ASCII characters and have the **.inpf** file extension. The data to be transmitted must be structured as a set of frames. Each frame will consist of 2 SYN characters, ASCII 22, one control character to indicate the length of data block followed by a maximum of 32 data characters. Assume that there are no trailing control characters. Every block transmitted must contain 32 data characters except possibly the case where the remainder of the set of characters cannot fill the frame. Each character will consist of 7 information bits and a parity bit. The two tables below illustrate the structure of a frame created to transmit the “*BCCA*” character string. The top table shows the frame contents in decimal and in the table at the bottom, the actual contents of the frame have been encoded with characters ‘0’ and ‘1’ to simulate the binary representation of the characters used.

2222	4	BCCA

0001011000010110	00000100	11000010010000110100001111000001

After the two pipes are created, the producer application uses one pipe to share all the encoded frames with the consumer. The producer also reads from another pipe all data processed by the consumer, deframes the data received, checks and removes parity bit, decodes the dada obtained, and it stores it in a file named *filename.done*.

The Consumer

Reads all encoded frames through the pipe used by the producer to transmit data, deframes the data received, checks and removes parity bit, decodes the dada obtained, converts all lowercase characters to uppercase, encodes the converted characters, adds the parity bit, creates all needed frames, stores the resulting information in a file ***(.chck)***, and uses a pipe to share the encoded result with the producer.

Important Tasks

The following are the fundamental tasks to be done for this project:

a. Create the C files needed for each of the three layers. Compile them and create your "encDec.h" file with the prototypes of all functions needed to provide the services defined by each of the three layers.
b. Create the consumer/producer application, as discussed in this document, that uses two shared pipes for communication. Your consumer/producer application must contain the statement #include "encDec.h"
c. Create and populate the *filename.inpf.* This file will contain all the original data to be shared between producer and consumer.
d. Create the *filename.binf*. This is the binary (0's and 1's) version of the original data. This will help you check the correctness of your conversion module. This file is created by the producer, shared with the consumer through the pipe, and processed by the consumer.
e. Create the *filename.outf*. This is the modified version of *filename.inpf* where all lower case letters have been replaced by upper case letters. The consumer is responsible for the creation of *filename.outf*.
f. Create the *filename.chck*. This file is the encoded version of *filename.outf* with parity bit and frame included.

What to Submit

a) Your solution must be uploaded to Blackboard.
b) Copies of all source files as well as their executables, and any data you used for testing your solution must be included.
c) You are to place all files that are related to your solution as well as your report document into a .zip file. Your .zip file must follow the format: *CSI 500 Project2 Your Name*.
d) The documentation associated with your solution must be typeset in MS Word. Marks will be deducted if you do not follow this requirement.

Your program should be developed using GNU versions of the C compiler. Your solution must use **Ordinary Pipes** (Lecture 03 – Interprocess Communication; slides 34-36). It should be layered, modularized, and well commented. The following is a tentative marking scheme and what is expected to be submitted for this assignment:

1. External Documentation (as many pages necessary to fulfill the requirements listed below.) including the following:
    a. Title page
    b. A table of contents
    c. [20%] System documentation
        i. A high-level data flow diagram for the system
        ii. A list of routines and their brief descriptions
        iii. Implementation details
    d. [5%] Test documentation
        i. How you tested your program
        ii. Test sets must include
            - input files *.inpf*; binary files *.binf*; and output files *.outf;* as well as the. *chck* and *.done* files.
            - You may use the quote below as one of your testing files. You may name it as *winVirus.inpf.*
    e. [5%] User documentation
        i. How to run your program
        ii. Describe parameter (if any)

2. Source Code
    a. [65%] Correctness
    b. [5%] Programming style
        i. Layering
        ii. Readability
        iii. Comments
        iv. Efficiency

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.

Code to Illustrate the *encDec.h* File.

encDec.h

```
int toUpper( char *inData );
```

toUpperClient.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]){
  int pid = fork();
  if( pid == 0 ){
    printf("Child!!!!\n");
    execl("toUpperService", "toUpperService",  "z", NULL );
  }
  else if( pid > 0){
    wait(NULL);
    printf("Parent!!!!\n");
  }
  else
    printf(" No fork this time! \n");
}
```

toUpperService.c

```
#include <stdio.h>
#include <ctype.h>
#include "encDec.h"

int main(int argc, char *argv[]){
   printf( "%c\n", toUpper( argv[1] ));
}

int toUpper( char *inData ){
  if( (int)*inData >= 'a' && (int)*inData <= 'z' )
    return( toupper( *inData ));
}
```