Project 2

Technical Documentation

Dawson Geist

SUNY University at Albany

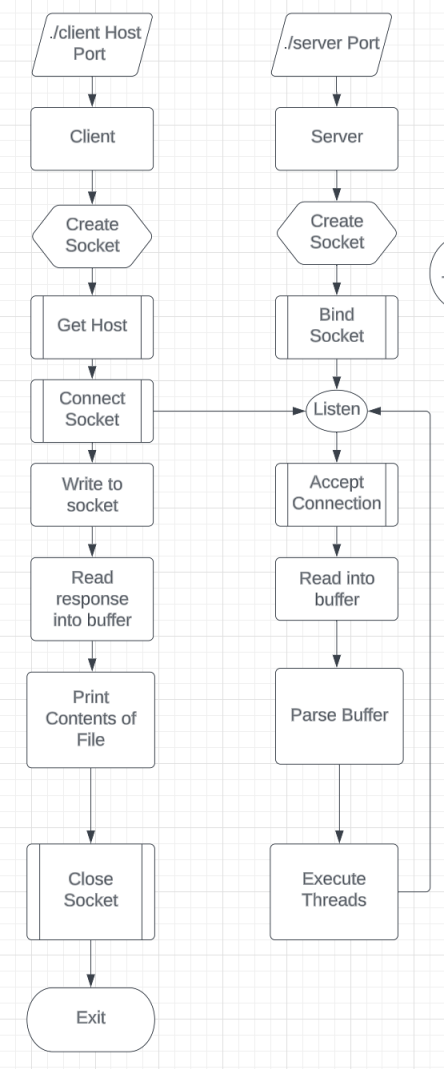
5/01/2021

Table of Contents

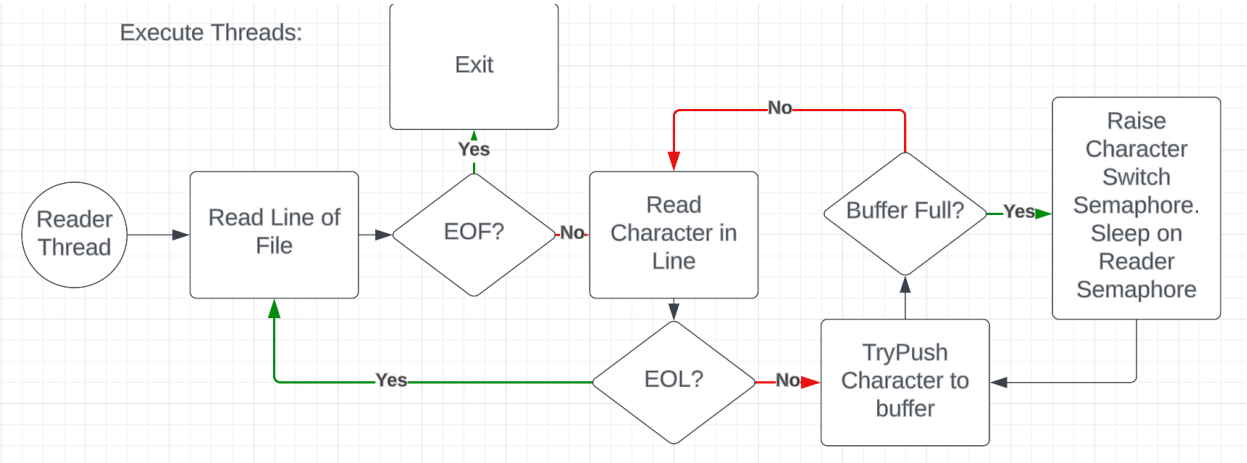
System Documentation

Data Flow Diagram

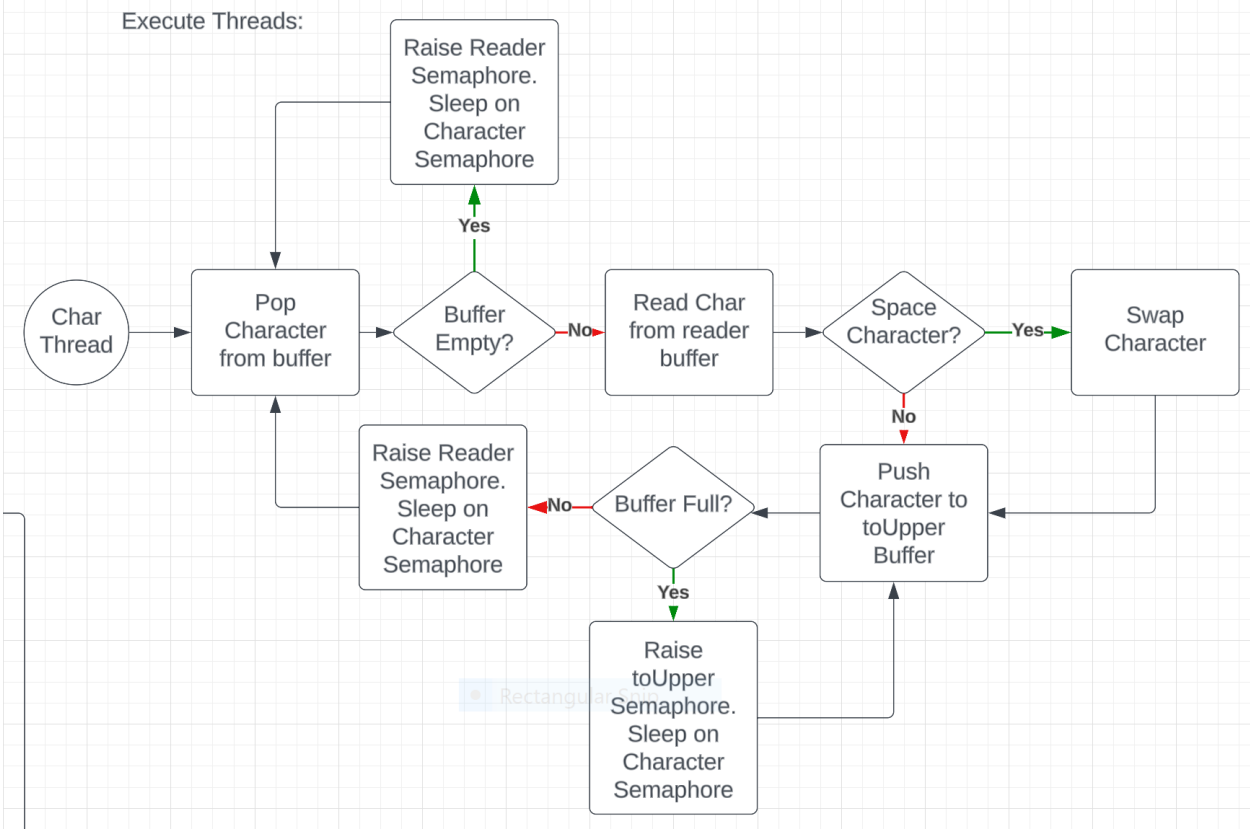
Client-Server



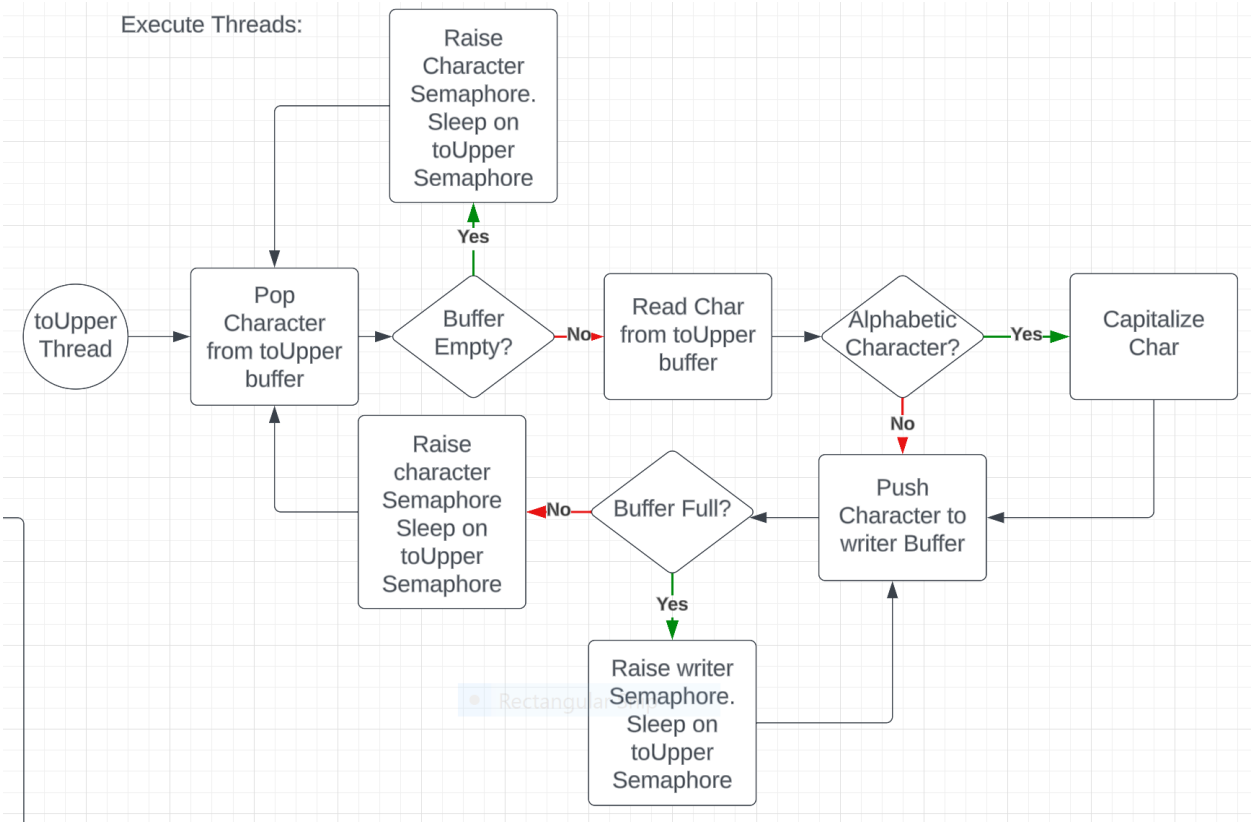
Reader Thread



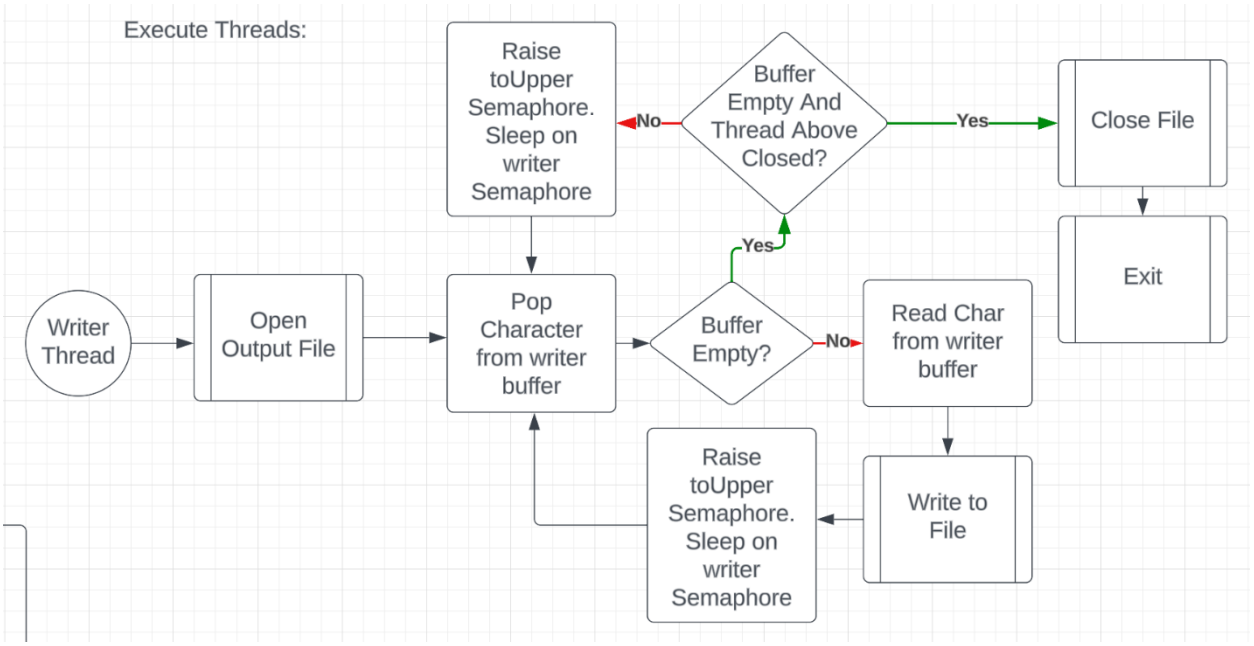
Character Thread



ToUpper Thread



Writer Thread



Functions / Semaphores – Producer.c

- int main():
    - Entry point for the program. Holds the Logic for the establishing the socket connection in addition to the creation and joining of various worker threads listed below. Also provides the communication of the output file name and location to the client.
- void error(const char *msg)
    - Prints the error message to perror. Used during the establishing of the socket connection
- void iterate(struct Buffer * buffer)
    - Test Method. Used to express the contents of the buffer as characters are added to them. The buffer is implemented using a linked list struct
- int tryPush(char val, struct Buffer * buffer)
    - Helper function. Used to add values to buffers. First it checks the buffer struct for its size/ # of elements. If the number of elements is greater than 10 this function will return -1, else, the function creates a new node with the appropriate value and adds it to the end of the buffer. Additionally the Buffer struct will update the Tail member of the struct so that it points to the newly created Node. On success, the count member of the buffer struct is incremented by 1.
- int pop(struct Buffer * buffer)
    - Helper function. Used to retrieve/ remove values from the link list / buffer. First it will check that the buffer length is greater than zero. When the buffer length is greater than zero the buffer head element is returned. In doing so, we update the buffer head so that it points to the next Node in the linked list. The old head node value is then returned to the user for further use. If the buffer length is zero the returned character will be null.
- void *readFile(void *filePath)
    - The “reader” thread function as it is written in the project description. This function opens the file passed as an argument and begins reading the file line by line. For every line that is in the file, the function goes character by character and trys to add the character to the reader buffer r_b. it will do this until the buffer is full. Once it is full, it will raise the character semaphore cR so that the characterReplace thread can run. Immediately after that it will wait on the reader semaphore r until it can try to add characters to the reader buffer. When it reads the last line in the buffer it will set a flag readerFinished to true which will tell the character Replace thread to no longer wait for readFile to raise its semaphore. After that it raises the character Replace semaphore cR telling the character Replace Thread that it is ok to run.
- void *charReplace(void * c)
    - The “character” thread function as it is written in the project description. This function starts by sleeping on the characterReplace semaphore cR. This is initially raised by the reader thread after it has completely filled the reader buffer. When the semaphore is raised the charReplace thread will loop continuously so long as the reader buffer length is greater than 0. While in this loop the charReplace thread pops a character from the

reader buffer. If the character is a whitespace that character is replaced by the argument character c. after it has the character value (either the replaced value or the original value) it will raise the reader semaphore r telling the reader thread it is ok to access the reader buffer. After that it will try to push the character value into the character buffer c_b. If it fails it will raise the to Upper semaphore tU and sleep on the character semaphore cR. This will repeat until the value is successfully added to the character buffer. Once the value is added to the character buffer the function checks to see if the readerFinished flag is false, if it is false it will sleep on the character Replace semaphore cR. Otherwise it will immediately return to the top of the loop where it will try to get the next character from the reader buffer. When the reader buffer is empty the character replace function will break out of its loop and set the charReplaceFinished flag to true. The last thing that is done before the function exits is raise the toUpper semaphore tU telling the toUpper thread it is ok to run.

- void *toUpper(void *fileLine)
    - The "toUpper" thread function as it is written in the project description. This function starts by sleeping on the toUpper semaphore tU. This is initially raised by the charReplace thread after it has completely filled the character buffer. When the semaphore is raised the charReplace thread will loop continuously so long as the character buffer length is greater than 0. While in this loop the toUpper thread pops a character from the character buffer. If the character is an Alphabetic character that character is capitalized, else, nothing happens. After it has the character value (either the capitalized value or the original value) it will raise the charReplace semaphore cR telling the charReplace thread it is ok to access the character buffer. After that it will try to push the character value into the toUpper buffer t_b. If it fails it will raise the writer semaphore w and sleep on the toUpper semaphore tU. This will repeat until the value is successfully added to the toUpper buffer. Once the value is added to the character buffer the function checks to see if the charReplaceFinished flag is false, if it is false it will sleep on the toUpper semaphore tU. Otherwise it will immediately return to the top of the loop where it will try to get the next character from the characterReplace buffer. When the characterReplace buffer is empty the toUpper function will break out of its loop and set the toUpperFinished flag to true. The last thing that is done before the function exits is raise the writer semaphore w telling the writer thread it is ok to run.
- void *writer(void *buffer)
    - The "writer" thread function as it is written in the project description. This function starts by opening an output file named "Output.txt". It is located in the same directory as the producer.c program. This file is created using the O_Trunc flag which clears any existing data before writing new data to it. After the file has been created and made ready for writing, the function sleeps on the writer semaphore w. This is initially raised by the toUpper thread after it has completely filled the toUpper buffer t_b. While the tUpper buffer length is greater than 0 the writer function will loop. During this loop the writer function will pop a value from the toUpper buffer t_b. Next it will add this character value to a character array named outBuffer. It will then raise the toUpper semaphore tU signaling to the toUpper thread that it is ok to run. After that, the writer thread checks to see if the toUpperFinished flag is false. If the flag is false it will sleep on

the writer semaphore w, else it will return to the top of the loop where it will again try to pop a value from the toUpper buffer. Once the toUpper buffer is depleted, the writer function will write the outBuffer to the outFile and then close the outFile. Finally, before exiting the function, the writer function will set the writerFinished flag to true.

- sem_t r
    - The reader semaphore. The primary purpose of this semaphore is to protect the reader buffer. There are two threads who have access to the reader buffer; reader thread, and the charReplace thread. Control is first given to the reader thread, where it will fill the reader buffer until it is at its maximum capacity. At this point the reader thread will sleep on r. This can only be broken by the charReplace thread. It will raise the reader semaphore r after it has successfully removed a character from the reader buffer
- sem_t cR;
    - The charReplace semaphore. The primary purpose of this semaphore is to protect the character buffer and the reader buffer. The character replace thread touches two buffers; reader buffer and character Replace buffer. This semaphore is initially raised by the reader thread which tells the characterReplace thread it is ok to read from the reader buffer. After that it will again sleep until it is again raised by either the reader thread or the toUpper thread
- sem_t tU;
    - The toUpper semaphore. The primary purpose of this semaphore is to protect the toUpper buffer and the charReplace buffer. The toUpper thread touches two buffers; charReplace buffer and toUpper buffer. This semaphore is initially raised by the charReplace thread which tells the toUpper thread it is ok to read from the charReplace buffer. After that it will again sleep until it is again raised by either the charReplace thread or the writer thread
- Sem_t w
    - The writer semaphore. The primary purpose of this semaphore is to protect the writer buffer. There are two threads who have access to the writer buffer; writer thread, and the toUpper thread. Control is first given to the toUpper thread, where it will fill the writer buffer until it is at its maximum capacity. At this point the writer thread will control the write buffer where it can pop values and add them to the out buffer. At this point if the toUpper thread is still running the writer thread will sleep on w. This can only be broken by the toUpper thread. It will raise the writer semaphore w after it has successfully added a character to the writer buffer

Functions / Semaphores - Consumer

- int main():
    - Entry point for the program. Holds the Logic for the establishing the socket connection Also provides the communication of the output file name and location from the server and then outputting the contents of the file to STDOUT.
- void error(const char *msg)
    - Prints the error message to perror. Used during the establishing of the socket connection

Implementation Details

The program was designed with simplicity in mind. It was in my opinion easiest to control the flow of control between threads if I had the threads pass control when either their respective buffers were full or when they removed values from their "neighboring" buffers. By doing this it implicitly creates a hierarchy to what threads can run when, and it usually follows the idea that:

1. The reader thread will always read until its buffer is full
2. The charReplace thread will come after the reader thread and slowly fill its buffer until it is maxed
3. When charReplace buffer is maxed the toUpper thread will come after and slowly build its buffer until it is maxed
4. When the toUpper buffer is maxed the writer thread will come after and continue syphoning off values until the toUpper buffer is completely empty

By using Boolean flags at the end of functions to signify the end of operation, the threads have a cascading shutoff where the highest level will execute until its' buffer is empty at which it will shutdown and the next highest level thread will execute until its' buffer is completely empty and so on and so forth until all the threads have completed.

Test Documentation

How it was Tested

The majority of the testing for this program came from printf() commands strategically placed throughout the code. These have been commented out in the code base but can be uncommented if needed.

Test Set

N/A

User Documentation

How to Run

After downloading the source code file the object file needs to be created.

1. Navigate to the directory where the source code is located.
2. Enter the following command
    a. gcc -o producer producer.c
    b. gcc -o consumer consumer.c
3. After the previous command executes you can run the program by entering the following command
    a. ./producer 54554
    b. ./consumer 127.0.0.1 54554

You are then free to enter any filename filepath and character to the consumer shell and the producer server will execute on the supplied file:

Parameters

Producer.c takes one parameter which is the port number that the socket should bind to.

Consumer.c takes two parameters which is the Hostname and Port number used by the server

Files

Intext.txt

source code

represents the part of

process that

contains the programming

language itself. you may

use a text editor to

write your source code file.

Output.txt

SOURCE#CODE

REPRESENTS#THE#PART#OF

PROCESS#THAT

CONTAINS#THE#PROGRAMMING

LANGUAGE#ITSELF.#YOU#MAY

USE#A#TEXT#EDITOR#TO

WRITE#YOUR#SOURCE#CODE#FILE.

Source Code

producer.c

```
/* Program: producer.c
* A simple TCP server using sockets.
* Server is executed before Client.
ICSI 412 Operating Systems Spring 2023
* Port number is to be passed as an argument.
*
* To test: Open a terminal window.
* At the prompt ($ is my prompt symbol) you may
* type the following as a test:
*
* $ ./producer 54554
* Run client by providing host and port
*
*
*/
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <fcntl.h> // for open
#include <unistd.h> // for close
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <fcntl.h> // for open
#include <unistd.h> // for close
#include <ctype.h>
```

```
void error(const char *msg)
{
    perror(msg);
    exit(1);
}

sem_t r;
sem_t cR;
sem_t tU;
sem_t w;

int retReader;
int retCharReplace;
int retToUpper;
int retWriter;

int readerFinished = 0;
int charReplaceFinished = 0;
int toUpperFinished = 0;
int writerFinished = 0;

void *readFile(void *);
void *charReplace(void *);
void *toUpper(void *);
void *writer(void *);

struct Node
{
    char value;
    struct Node * next;
};

struct Buffer
{
    int count;
    char b[10];
    struct Node * head;
    struct Node * tail;
}r_b,c_b,t_b;

void iterate(struct Buffer * buffer)
{
    struct Node * cur = buffer->head;
    while(cur != NULL)
    {
```

```
        printf("Node val: %c\n",(char)cur->value);
        cur = cur->next;
    }
    printf("\n");
}

int tryPush(char val, struct Buffer * buffer)
{
    int success = -1;
    if(buffer->count < 10)
    {
        //printf("ADDING %c TO BUFFER\n",val);
        //int i = 0;
        //while(buffer->b[i] != '\0')
        //{
        //  i++;
        //}
        //buffer->b[i] = val;
        //buffer->count++;
        //success = 1;
        // Linked list
        struct Node * newChar = (struct Node*)malloc(sizeof(struct Node));
        newChar->value = val;
        newChar->next = NULL;
        if(buffer->head == NULL && buffer->tail == NULL)
        {
            buffer->head = newChar;
            buffer->tail = newChar;
        }
        else
        {
            struct Node * curTail = buffer->tail;
            curTail->next = newChar;
            buffer->tail = newChar;
        }
        //iterate(buffer);
        buffer->count++;
        success = 1;
    }
    //printf("BUFFER CONTENTS: \n");
    for(int i = 0; i < 10; i++)
    {
        //printf("[%d] %c\n", i, buffer->b[i]);
    }
    return success;
```

```
}

int pop(struct Buffer * buffer)
{
    int val;
    if(buffer->count != 0)
    {
        //printf("REMOVING FIRST VALUE FROM BUFFER\n");
        //val = buffer->b[0];
        //for(int i = 0; i < 9; i++)
        //{
        //  char next = buffer->b[i+1];
        //  buffer->b[i] = next;
        //}
        //buffer->b[9] = '\0';
        //buffer->count--;
        //Linked List
        val = buffer->head->value;
        struct Node * newHead = buffer->head->next;
        //free(buffer->head);
        buffer->head = newHead;
        //iterate(buffer);
        buffer->count--;
    }
    //printf("BUFFER CONTENTS: \n");
    for(int i = 0; i < 10; i++)
    {
        //printf("[%d] %c\n", i, buffer->b[i]);
    }
    //printf("VAL: %c\n", (char)val);
    return val;
}

void *readFile(void *filePath)
{
    int input;
    FILE *fp = fopen(filePath, "r");

    char line[255];

    //printf("r:BEGIN READING FILE\n");

    while(fgets(line, sizeof(line), fp) != NULL)
    {
        //printf("%s\n", line);
```

```
        int i = 0;
        while(line[i] != '\0')
        {
            //printf("r:ADDING CHAR %c TO BUFFER r_b\n",line[i]);
            //If we try to add to the buffer but the buffer is full
            while((tryPush(line[i], &r_b) != 1))
            {
                //printf("r:BUFFER r_b FULL... TELLING charReplace OK to RUN\n");
                //Tell the charReplace Thread it is ok to grab from the queue
                sem_post(&cR);
                //printf("r:WAITING for OK to RUN\n");
                //And wait for charReplace to tell us its ok to resume
                sem_wait(&r);
                //charReplace says its safe to try again so we block charReplace
so that the buffer is safe.
            }
            //printf("r:CHAR %c ADDED TO BUFFER r_b\n",line[i]);
            i++;
        }
    }

    //printf("r:EXITING\n");
    //While cR is still blocked raise the flag saying we are finished
    readerFinished = 1;
    //Tell the charReplace Thread it is ok to grab from the queue
    sem_post(&cR);
}

void *charReplace(void * c)
{
    //printf("cR:   WAITING FOR OK TO BEGIN\n");
    //Wait for permission to run because readFile needs to begin filling the
buffer
    sem_wait(&cR);
    //printf("cR:   RUNNING\n");
    while(r_b.count > 0)
    {
        //Grab character from buffer
        //printf("cR:   GRABBING VAL FROM BUFFER r_b\n");
        char val = (char)pop(&r_b);
        if(val == ' ')
        {
            val = (char)c;
        }
        //printf("cR:   TELLING reader OK to RUN\n");
```

```
        // Tell the reader thread it is ok to run
        sem_post(&r);
        //printf("cR:   ADDING %c TO BUFFER c_b\n", val);
        //Push the new character into the charReplace Buffer
        while((tryPush(val, &c_b) != 1))
        {
            //printf("cR:   c_b FULL\n");
            //printf("cR:   TELLING toUpper OK to RUN\n");
            // Tell the toUpper thread it is ok to run
            sem_post(&tU);
            //printf("cR:   WAITING FOR OK TO RESUME\n");
            //Wait to try again
            sem_wait(&cR);
        }
        //printf("cR:   %c ADDED TO BUFFER c_b\n", val);
        // if the reader finished executing only wait on the writer
        //if(readerFinished == 0 )
        if(readerFinished == 0)
        {
            //printf("cR:   WAITING FOR OK TO RESUME\n");
            sem_wait(&cR);
        }
    }
    //printf("cR:   EXITING\n");
    charReplaceFinished = 1;
    sem_post(&tU);
}

void *toUpper(void *fileLine)
{
    //printf("tU:       WAITING FOR OK TO BEGIN\n");
    //wait until someone says we can run
    sem_wait(&tU);
    while(c_b.count > 0)
    {
        //printf("tU:       GRABBING VAL FROM BUFFER c_b\n");
        //Grab character from buffer
        char val = (char)pop(&c_b);
        if(isalpha(val))
        {
            val = toupper(val);
        }
        //printf("tU:       TELLING charReplace OK to RUN\n");
        // Tell the characterReplace thread it is ok to run
        sem_post(&cR);
```

```
        //printf("tU:       ADDING %c to BUFFER t_b\n",val);
        //Push the new character into the charReplace Buffer
        while((tryPush(val, &t_b) != 1))
        {
            //printf("tU:       BUFFER t_b FULL\n");
            //printf("tU:       TELLING writer OK to RUN\n");
            // Tell the writer thread it is ok to run
            sem_post(&w);
            //printf("tU:       WAITING FOR OK TO RESUME\n");
            sem_wait(&tU);
        }
        //printf("tU:       %c ADDED to BUFFER t_b\n",val);
        if(charReplaceFinished == 0)
        {
            //printf("tU:       WAITING FOR OK TO RESUME\n");
            sem_wait(&tU);
        }
    }
    //printf("tU:       EXITING\n");
    toUpperFinished = 1;
    // Tell the writer thread it is ok to run
    sem_post(&w);
}

void *writer(void *buffer)
{

    //File Setup
    int outFile;
    outFile = open("Output.txt", O_TRUNC | O_CREAT | O_RDWR, S_IRWXU);
    //File Buffer Setup
    char outBuffer[512];
    int i = 0;
    //printf("w:            WAITING FOR OK TO BEGIN\n");
    //wait until someone says we can run
    sem_wait(&w);
    while(t_b.count > 0)
    {
        //printf("w:            GRABBING VAL FROM BUFFER c_b\n");
        //Grab character from buffer
        char val = (char)pop(&t_b);
        outBuffer[i++] = val;
        //printf("w:            TELLING toUpper OK to RUN\n");
        // Tell the toUpper thread it is ok to run
        sem_post(&tU);
```

```
        if(toUpperFinished == 0)
        {
            //printf("w:            WAITING FOR OK TO RESUME\n");
            // Wait until its ok to run again
            sem_wait(&w);
        }
    }

    // Before we exit write the contents of our outbuffer to the outfile
    write(outFile, outBuffer, strlen(outBuffer));
    close(outFile);
    //printf("w:            EXITING\n");
    writerFinished = 1;
}

int main(int argc, char *argv[])
{
    retReader = sem_init(&r, 0, 0);
    retCharReplace = sem_init(&cR, 0, 0);
    retToUpper = sem_init(&tU, 0, 0);
    retWriter = sem_init(&w, 0, 0);

    pthread_t thR, thCR, thTU, thW;


    int sockfd, newsockfd, portno;
    int end = 0;
    socklen_t clilen;
    char buffer[256];
    char filename[32];
    char fileLocation[256];
    char character;
    struct sockaddr_in serv_addr, cli_addr;
    int n;
    if (argc < 2) {
        fprintf(stderr,"ERROR, not enough arguments\n");
        exit(1);
    }
    fprintf(stdout, "Run client by providing host and port\n");
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    bzero((char *) &serv_addr, sizeof(serv_addr));
    portno = atoi(argv[1]);
    serv_addr.sin_family = AF_INET;
```

```
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);
    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
        error("ERROR on binding");
    listen(sockfd,5);
    clilen = sizeof(cli_addr);
    while(!end)
    {
        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
        if (newsockfd < 0)
            error("ERROR on accept");
        bzero(buffer,256);
        n = read(newsockfd,buffer,255);
        if (n < 0)
            error("ERROR reading from socket");
        //printf("Here is the message: %s\n",buffer);

        // Parse the client sent string
        int i = 0;
        //parse command
        while(i < strlen(buffer) && buffer[i] != ' ')
        {
            filename[i] = buffer[i];
            i++;
        }
        filename[i] = '\0';
        //printf("FileName: %s\n", filename);
        i++;
        //printf("Begin Parsing Arguments\n");
        int j = 0;
        //parse arguments
        while(i < strlen(buffer) && (buffer[i] != ' '))
        {
            //printf("Adding Character: %c\n", buffer[i]);
            fileLocation[j] = buffer[i];
            i++;
            j++;
        }
        fileLocation[j] = '\0';
        j = 0;


        //printf("Begin Parsing 2 Argument\n");
        i++;
        //parse arguments
```

```
        while(i < strlen(buffer) && buffer[i] != ' ' && buffer[i] != '\n')
        {
            //printf("Adding Character: %c\n", buffer[i]);
            character = buffer[i];
            i++;
            j++;
        }
        // Parse the client sent string
        //printf("Filename: %s\n", filename);
        //printf("File Location: %s\n", fileLocation);
        //printf("Char: %c\n", character);

        // Thread Stuff

        // Process File
        pthread_create( &thR, NULL, readFile, (void*)fileLocation);
        pthread_create( &thCR, NULL, charReplace, (void*)character);
        pthread_create( &thTU, NULL, toUpper, NULL);
        pthread_create( &thW, NULL, writer, NULL);

        pthread_join(thR,NULL);
        pthread_join(thCR, NULL);
        pthread_join(thTU,NULL);
        pthread_join(thW, NULL);

        //readFile("testInput.txt");
        //tryPush('H',&r_b);
        //tryPush('O',&r_b);
        //tryPush('T',&r_b);
        //tryPush('E',&r_b);
        //tryPush('L',&r_b);
        //char val;
        //val = (char)pop(&r_b);
        //End Thread Stuff

        bzero(filename,32);
        bzero(fileLocation,256);
        character = '\0';

        //Write to client
        n = write(newsockfd,"Output.txt Output.txt",21);
        if (n < 0)
            error("ERROR writing to socket");
    }
```

```
    close(newsockfd);
    close(sockfd);
    return 0;
}
```

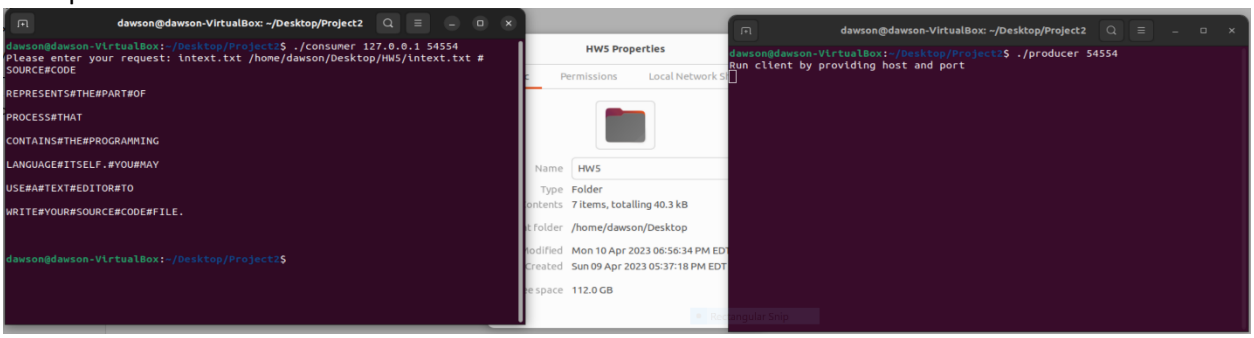
consumer.c

```
/*
* Simple client to work with server.c program.
* Host name and port used by server are to be
* passed as arguments.
*
* To test: Open a terminal window.
* At prompt ($ is my prompt symbol) you may
* type the following as a test:
*
* $./consumer 127.0.0.1 54554
* Please enter the message: Programming with sockets is fun!
* I got your message
* $
*
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
void error(const char *msg)
{
    perror(msg);
    exit(0);
}
int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;
    char buffer[256];
    char filename[32];
    char fileLocation[64];
```

```
    if (argc < 3) {
        fprintf(stderr,"usage %s hostname port\n", argv[0]);
        exit(0);
    }
    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr, (char *) &serv_addr.sin_addr.s_addr, server-
>h_length);
    serv_addr.sin_port = htons(portno);
    if (connect(sockfd, (struct sockaddr *)&serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR connecting");
    printf("Please enter your request: ");
    bzero(buffer,256);
    fgets(buffer,255,stdin);
    n = write(sockfd,buffer,strlen(buffer));
    if (n < 0)
        error("ERROR writing to socket");
    bzero(buffer,256);
    n = read(sockfd,buffer,255);
    if (n < 0)
        error("ERROR reading from socket");
    else
    {
        // Parse the client sent string
        int i = 0;
        //parse command
        while(i < strlen(buffer) && buffer[i] != ' ')
        {
            filename[i] = buffer[i];
            i++;
        }
        filename[i] = '\0';
        //printf("FileName: %s\n", filename);
        i++;
        //printf("Begin Parsing Arguments\n");
        int j = 0;
```

```
        //parse arguments
        while(i < strlen(buffer) && (buffer[i] != ' '))
        {
            //printf("Adding Character: %c\n", buffer[i]);
            fileLocation[j] = buffer[i];
            i++;
            j++;
        }
        fileLocation[j] = '\0';
        j = 0;
        // Parse the client sent string
        //printf("Filename: %s\n", filename);
        //printf("File Location: %s\n", fileLocation);
    }

    FILE *fp = fopen(fileLocation, "r");

    char line[255];

    //printf("r:BEGIN READING FILE\n");

    while(fgets(line, sizeof(line), fp) != NULL)
    {
        printf("%s\n", line);
    }

    close(sockfd);
    return 0;
}
```

Output


dawson@dawson-VirtualBox: ~/Desktop/Project2
dawson@dawson-VirtualBox:~/Desktop/Project2$ ./consumer 127.0.0.1 54554
Please enter your request: intext.txt /home/dawson/Desktop/HW5/intext.txt #
SOURCE#CODE
REPRESENTS#THE#PART#OF
PROCESS#THAT
CONTAINS#THE#PROGRAMMING
LANGUAGE#ITSELF.#YOU#MAY
USE#A#TEXT#EDITOR#TO
WRITE#YOUR#SOURCE#CODE#FILE.
dawson@dawson-VirtualBox:~/Desktop/Project2$
HW5 Properties
Permissions
Name
HW5
Type
Folder
7 items, totalling 40.3 kB
/home/dawson/Desktop
Mon 10 Apr 2023 06:56:34 PM EDT
Sun 09 Apr 2023 05:37:18 PM EDT
112.0 GB
dawson@dawson-VirtualBox: ~/Desktop/Project2
dawson@dawson-VirtualBox:~/Desktop/Project2$ ./producer 54554
Run client by providing host and port