Documentation for Project2
Name: H9

Teacher: Guanghui Chang, Jackson Marques de Carvalho
Course Title: Operating system
Data: 2023.06.06

Directory

1. System documentation

1.1 Data flow diagram

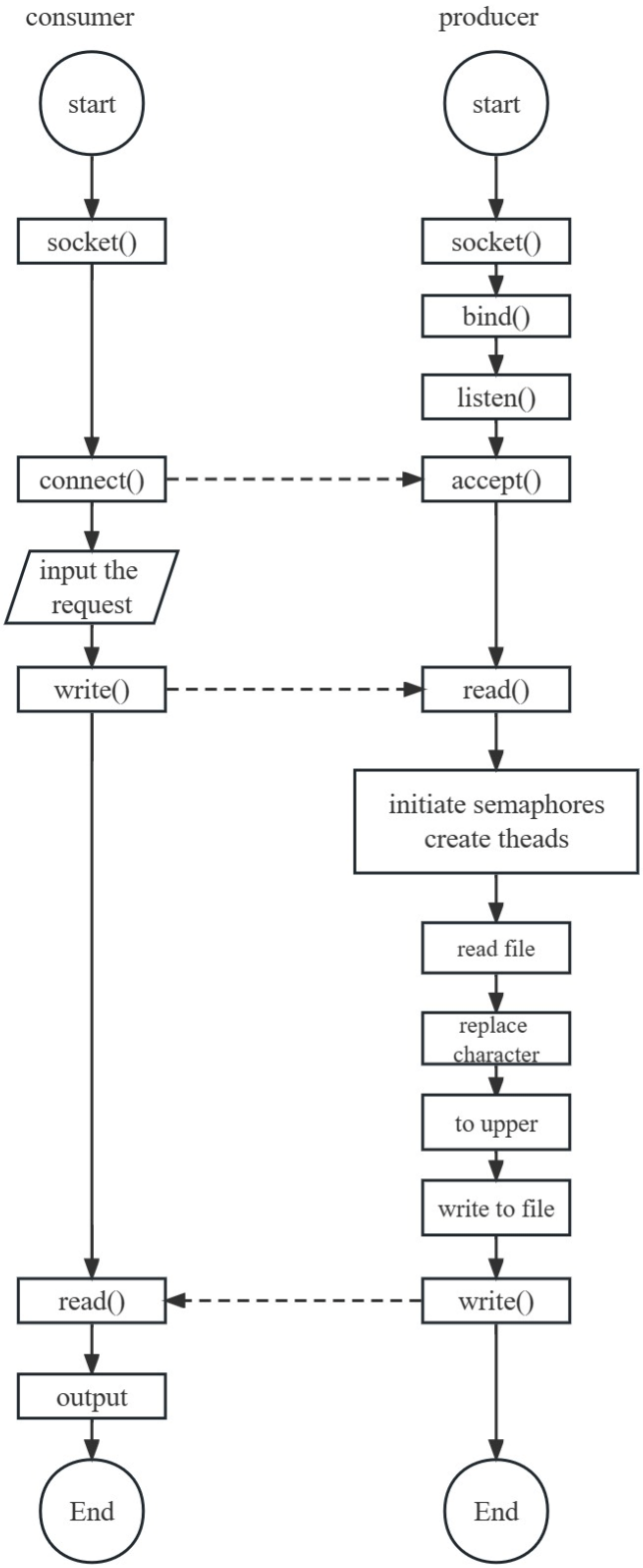

Figure 1 The whole program

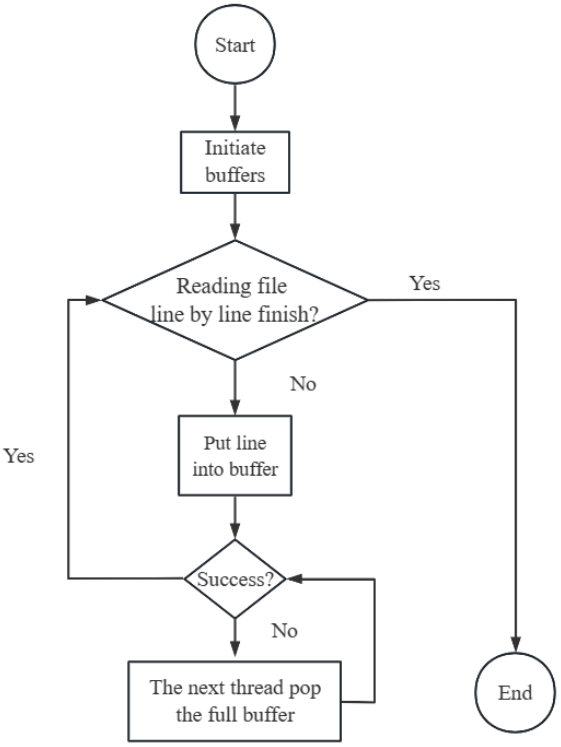

Figure 2 reader thread

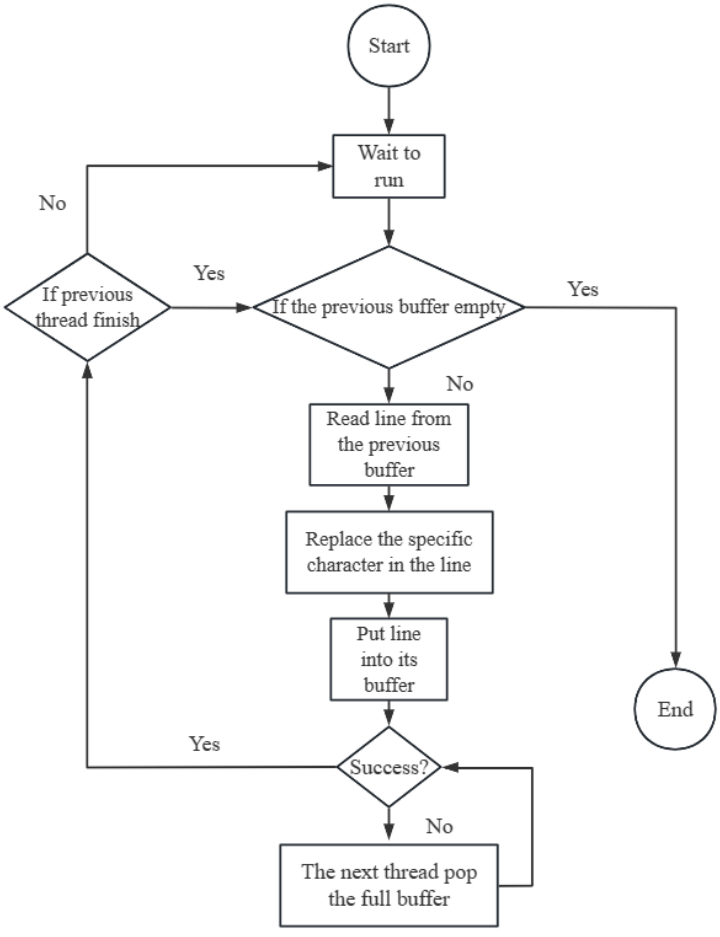

Figure 3 replace character thread

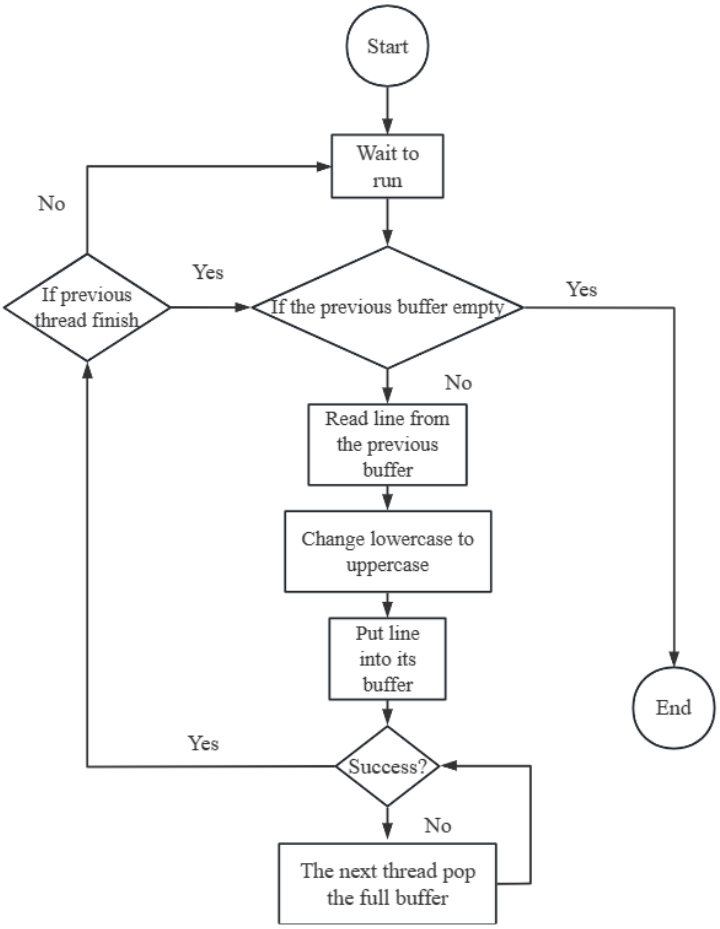

Figure 4 upper thread

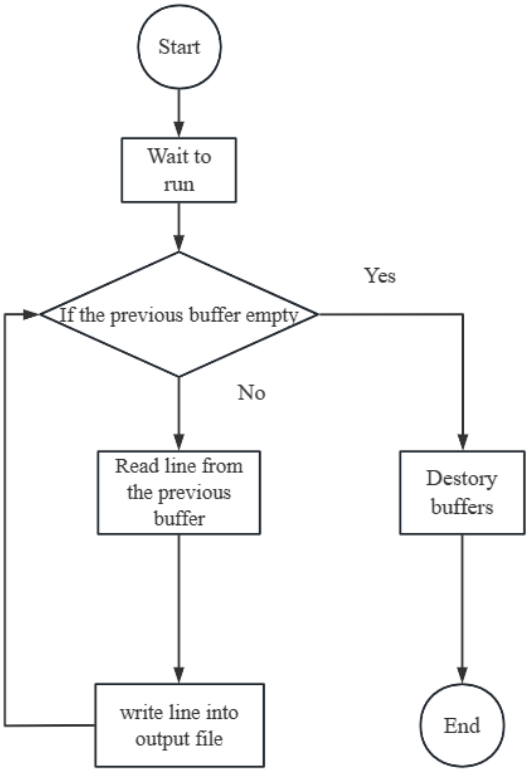

Figure 5 writer thread

1.2 Routines

Producer:

1. struct Node: Contains a string and a pointer to the next node
2. struct Buffer: A data struct for queue. Contains the size of the buffer, a pointer to the head node and a pointer to the tail node
3. int tryPush(char* value, struct Buffer* buffer): Push a new node to the buffer. The new node will be the tail. The string of the new node is the value and the max size of the buffer is 10. If success, return 1, else return -1.
4. char* pop(struct Buffer* buffer): Pop the head of the buffer.
5. void error(const char *msg): Print the information of the error and interrupt the program.
6. void *readFile(void *filePath): Read the contents of the file by line to its buffer. When the buffer is full, pop the buffer by executing the next thread.
7. void *charReplace(void* c): Execution starts when the execution of the current thread is complete or when the buffer of the current process is full. Reads the contents of the previous process buffer by line, replaces the spaces in the buffer with the specified characters, and writes them to its buffer when finished. When the buffer is full, pop the buffer by executing the next thread.
8. void *toUpper(): Execution starts when the execution of the current thread is complete or when the buffer of the current process is full. Reads the contents of the previous process buffer on a per-line basis and replaces the lower case in the buffer with upper case, writing to its buffer when finished. When the buffer is full, pop the buffer by executing the next thread.
9. void *writer(): Execution starts when the execution of the current thread is complete or when the buffer of the current process is full. Write the contents of the previous thread's buffer to the output file.
10. int main(int argc, char *argv[]): First create a socket, then listen, accept the consumer's request, read the file information provided by the consumer, create a thread to process the file information, and send the output file information to the consumer when finished.

Consumer:

1. void error(const char *msg): Print the information of the error and interrupt the program.
2. int main(int argc, char *argv[]): First create a socket, then connect to the server, send the input file information to the producer, receive the output file information from the producer, and output the file content

1.3 List of all semaphores

- sem_t read_sem: Semaphore for read thread. If the buffer of read thread is full, it will wait.
- sem_t change_sem: Semaphore for for replace character thread. Controls when the replace thread will run, and will go to wait when his buffer is full.
- sem_t upper_sem: Semaphore for toUpper thread. Controls when the upper thread will run, and will go to wait when his buffer is full.
- sem_t write_sem: Semaphore for write thread. Controls when the writer thread will

run.

1.4 Implementation details.

1. Create data struct Node and Buffer to present a string queue.

```
struct Node
{
        char value[256];
        struct Node* next;
};

/**
 * Description: struct for buffer
 * @ haed The head node
 * @ tail The tail node
 */
struct Buffer
{
        int size;
        struct Node* head;
        struct Node* tail;
};
```

Figure 6 Detail1

2. Use semaphore to control the order of thread execution

```
// Read context one line at a time from file
while(fgets(line, sizeof(line), fp) != NULL)
{
        //printf("%d ", line[i]);
        // When buffer is full
        while((tryPush(line, read_buffer) != 1))
        {
                // Character replace thread grab from the queue
                sem_post(&change_sem);

                // Wait for character replace thread finish grabing
                sem_wait(&read_sem);
        }
```

Figure 7 Detail2

3. Use flag to record if thread finished

```
// Finish reader thread
//printf("read arrive first\n");
readerFinished = 1;
sem_post(&change_sem);
```

Figure 8 Detail3

2. Test documentation

2.1 Test the whole program

Input file:

Figure 9 test file

Producer: ./producer 1024

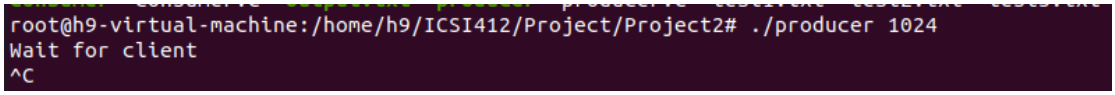

Figure 10 producer

Consumer: ./consumer 127.0.0.1 1024

Figure 11 consumer

2.2 Bugs(finished)

Inability to properly control the order of threads running- Adding semaphores and thread completion flags

3. User documentation

3.1 How to run

1. Execute in shell: ./producer port (port from 1024-65535)
2. Shell show: Wait for client
3. Execute in another shell: ./consumer 127.0.0.1 port (127.0.0.1- host loopback address, if you have more than 2 hosts then you can use other ip)
4. Shell show: Input the request:
5. Input in consumer shell: filename fileLocation character

Example: test.txt /home/h9/ICSI412/Project/Project2/test.txt X

3.2 Paraments

Producer:

argv[1]: port

Consumer:

argv[0]: ip address

argv[1]: port