HW5 REPORT

In my producer-consumer implementation, producer does not wait till the buffer is completely empty, when there is a empty place in the buffer, then producer is able to fill that place. Consumers do not wait as well, if an item exists in the buffer then a consumer locks the mutex, gets the values from the buffer and unlocks the mutex to let other consumers and producer use the buffer.

There the producer critical section code is below. When writing to buffer, firstly there should be empty place in buffer also no consumers should be using the buffer at that time. KillSignal is raised when SIGINT received, if that happens, then producer should close the files and directory that it opened, and leave immediately.

```
pthread_mutex_lock(&Mutex_m);
// if there is no empty slot in the buffer, keep waiting till an item is consumed
while ((bufIndex = FindEmptyInBuffer(Buffer, bufferSize)) == -1)
{
    pthread_cond_wait(&Cond_empty, &Mutex_m);
}
if (KillSignal == 1)
{
    pthread_cond_signal(&Cond_full);
    pthread_mutex_unlock(&Mutex_m);

    if (fd_dest > 0)
        close(fd_dest);
    if (fd_src > 0)
        close(fd_src);

    closedir(dir);
    return NULL;
}
// place the current file informations in the buffer.
strcpy(Buffer[bufIndex].fileName, entry->d_name);
Buffer[bufIndex].destFd = fd_dest;
Buffer[bufIndex].sourceFd = fd_src;

pthread_cond_signal(&Cond_full);
pthread_mutex_unlock(&Mutex_m);
```

There the consumer code is below, consumer waits till the buffer is available like producer does, also when kill signal raised, consumer signals other consumers by signaling condition variable Cond_full. Like a domino, each thread is closed one by one.

```
pthread_mutex_lock(&Mutex_m);
while ((bufIndex = FindFullInBuffer(Buffer, bufferSize)) == -1)
{
    // producer stopped producing or kill signal received
    if (Done == 1 || KillSignal == 1)
    {
        // don't let others wait
        pthread_cond_signal(&Cond_full);
        pthread_mutex_unlock(&Mutex_m);
        return NULL;
    }
    pthread_cond_wait(&Cond_full, &Mutex_m);
}
FileInfoBuffer chosenFile = Buffer[bufIndex];
Buffer[bufIndex].sourceFd = -1; // make it empty for producer

pthread_cond_signal(&Cond_empty);
pthread_mutex_unlock(&Mutex_m);
```

This is from consumer code, another critical section that has to be protected. Printf's are protected by locking Mutex_write_stdout as requested in the pdf.

```
// Protect writing to stdout and totalByte count and file count at the same time no need for another mutex
pthread_mutex_lock(&Mutex_write_stdout);
printf("consumer|file:%s has been copied successfully. %d bytes copied.\n", chosenFile.fileName, fileSize);
TotalByteCopied += fileSize;
FilesSucceed++;
close(chosenFile.sourceFd);
close(chosenFile.destFd);
pthread_mutex_unlock(&Mutex_write_stdout);
```

In pdf it is said that producer function takes array of at least 2 entries as parameter, instead of array, I used struct to be more readable. The logic is the same.

DirCount is used to understand all the directories are traversed recursively and after the last directory is traversed, done flag raised

```
typedef struct Directories
{
    char source[256];
    char destination[256];
    int bufferSize;
    int dirCount;
} Directories;
```

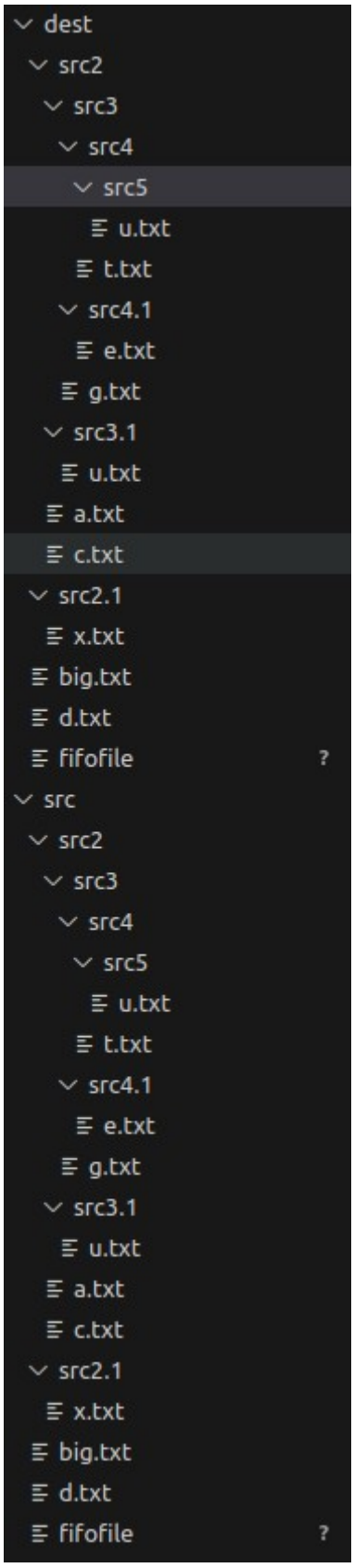
The code where dirCount is used.

```
// when all the sub directories are traversed, main directory w
if (direcs->dirCount == 0)
{
    // protect done flag because consumer may get effected badl
    pthread_mutex_lock(&Mutex_m);
    Done = 1;
    // awake waiting threads
    pthread_cond_broadcast(&Cond_full);

    // also protect print to stdout as it is requested in pdf
    pthread_mutex_lock(&Mutex_write_stdout);
    printf("Producer done.\n");

    pthread_mutex_unlock(&Mutex_write_stdout);
    pthread_mutex_unlock(&Mutex_m);
}
```

This is the output of the program. I haven’t detected any memory leak.

```
==8714== Command: ./main 5 5 src dest
==8714==
FIFO file:fifofile has been copied successfully.
Producer done.
consumer|file:c.txt has been copied successfully. 196690 bytes copied.
consumer|file:a.txt has been copied successfully. 823960 bytes copied.
consumer|file:u.txt has been copied successfully. 925650 bytes copied.
consumer|file:u.txt has been copied successfully. 925650 bytes copied.
consumer|file:e.txt has been copied successfully. 1936640 bytes copied.
consumer|file:x.txt has been copied successfully. 685153 bytes copied.
consumer|file:g.txt has been copied successfully. 1728000 bytes copied.
consumer|file:t.txt has been copied successfully. 7319520 bytes copied.
consumer|file:d.txt has been copied successfully. 7724971 bytes copied.
consumer|file:big.txt has been copied successfully. 251686224 bytes copied.
Directory copied in 4 seconds.
Regular Files: 10 succeed, 0 failed.
FIFO Files: 1 succeed, 0 failed.
Directories: 7 succeed, 0 failed.
Total of 273952458 bytes have been copied.
==8714==
==8714== HEAP SUMMARY:
==8714==     in use at exit: 0 bytes in 0 blocks
==8714==   total heap usage: 27 allocs, 27 frees, 594,704 bytes allocated
==8714==
==8714== All heap blocks were freed -- no leaks are possible
==8714==
==8714== For lists of detected and suppressed errors, rerun with: -s
==8714== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Directory hierarchy



This is how I handled the signal SIGINT. Make KillSignal 1 to notify all the threads. Clear the buffer and close open files, and then broadcast the condition variables to make sure threads are not stuck.

```
void MySignalHandler(int signo)
{
    switch (signo)
    {
    case SIGINT:
        KillSignal = 1; // killSignal raised, threads will unc
        printf("SIGINT received, terminating...\n");
        // clear the buffer and close all opened file descript
        if (Buffer != NULL)
        {
            for (int i = 0; i < BUFFER_SIZE; i++)
            {
                if (Buffer[i].sourceFd > 0)
                    close(Buffer[i].sourceFd);

                Buffer[i].sourceFd = -1;

                if (Buffer[i].destFd > 0)
                    close(Buffer[i].destFd);
            }
        }
        OpenedFileCount = 0;

        // make sure that threads are not blocked
        pthread_cond_broadcast(&Cond_empty);
        pthread_cond_broadcast(&Cond_full);
        pthread_cond_signal(&Cond_fd_count);
        return;
        break;

    default:
        break;
    }
}
```

To not to exceed per process fd limit, I created a global variable called

```
int OpenedFileCount = 0;
```

Then, when producer opened a file for reading and writing separately, opened file count is increased by 2. But this section should be protected so I protected it with mutexes.

```
// if no more  fd can be opened, then wait for a consumer to finish its job.
pthread_mutex_lock(&Mutex_open_file);
while (OpenedFileCount > MAX_OPEN_FILE_COUNT)
{
    pthread_cond_wait(&Cond_fd_count, &Mutex_open_file);
}
pthread_mutex_unlock(&Mutex_open_file);
```

```
// 2 more files are opened, increase the count
pthread_mutex_lock(&Mutex_open_file);
OpenedFileCount += 2;
pthread_mutex_unlock(&Mutex_open_file);
```

Also, consumer decreases OpenedFileCount when it closed the files.

```
// consumer is done with the files, decrease the opened file count
pthread_mutex_lock(&Mutex_open_file);
OpenedFileCount -= 2;
pthread_cond_signal(&Cond_fd_count); // signal for producer to produce
pthread_mutex_unlock(&Mutex_open_file);
```

Lastly “Experiment with different buffer sizes and different number of consumer threads” is requested in the pdf.

Big sized files are used when testing the program. If the files are small sized, then it finishes immediately.

Buffer size 1, consumer threads 1 = 16 seconds, 20 seconds, 21 seconds.

Buffer size 10, consumer threads 1 = 23 seconds, 25 seconds,28 seconds

Buffer size 10, consumer threads 10 = 23 seconds,24 seconds, 20seconds, 41seconds(once)

Buffer size 10 consumer threads 100 = 28 seconds,22 seconds, 23 seconds,

buffer size 100 consumer threads 100 = 21 seconds,28 seconds, 20 seconds

I am getting the best result when the buffer size is 1 and consumer thread pool size is 1 but this is not exactly correct for all trials, because when the buffer size or thread pool size is changed, sometimes time is not changing proportionally. I think this is because of my implementation. I used array of struct FileInfoBuffer to store the fds and filename, so when the buffer size increased, consumers should search in an array and this causes to performance loss (not too much in my case) also producer is able to fill the buffer whenever the buffer has empty place so producer fills, immediately consumer consumes without searching anything in the buffer. But if I implemented producer to produce only when the buffer is completely empty, then results could be different.