CMPSC 472 Project 1

Description:

This project is a file processing program designed to count the occurrences of a particular word or set of words within the contents of a file text. Each file is processed using separate processes which have 4 separate threads by default, but this can be configured by the user. It also incorporates interprocess communication (IPC) for communication between the child and parent processes to obtain the end result.

Structure of the Code:

Definitions

Four main constants are defined at the top of the code after the library inclusions, these include the `BUFFER_SIZE` (bites processed at a time while reading text file), `THREAD_COUNT` (number of threads used to process each file), `files[]` (array of file paths for processing), and `word` (word to search for).

Data Structures

This code includes one custom data structure called `ThreadArgs`, which contains the arguments for each thread responsible for processing a certain portion of the file. This structure includes the file path, word to search for, start byte index, end byte index, and count variable for word occurrences.

```
typedef struct {
    const char *path;
    const char *word;
    int start;
    int end;
    int count;
} ThreadArgs;
```

File Processing

The `processFile` method encapsulates most of the file processing. The method starts by preparing each file by determining its size, then dividing it into a number of parts corresponding to `THREAD_COUNT` such that each thread gets an equal part of the file to process. It then creates the threads, running `threadFunc` on each of them, and pipes their output back to the main function.

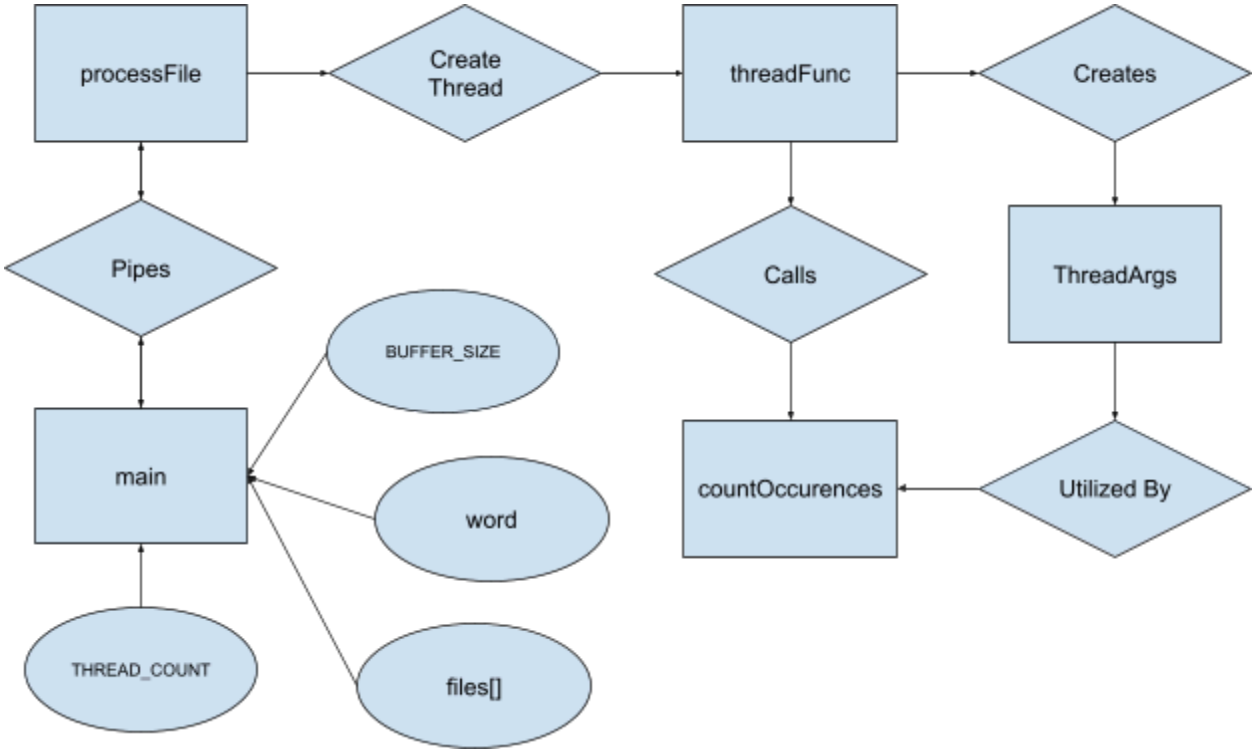
Thread Function

The `threadFunc` method creates a unique instance of `ThreadArgs` depending on its input and uses the struct to process and store the output of the `countOccurences` function.

Occurrence Counting

The `countOccurences` function opens the portion of the file it was allotted and counts all instances of the given word in its text, processing chunk by chunk based on the buffer size. It then returns the count back to the `processFile` method, which uses a pipe to write it back to main.

Diagram:



How to Run the Code:

Step 1: Open or copy the file's contents into Google Colab.

Step 2: Upload the text files to the root directory of the colab instance.

Step 3: Run the code in Colab.

Alternatively, use this link which can also be found in the repository's README:

https://colab.research.google.com/drive/18vNwJyXsM2MMvNwi4c4bhQcnAqgiJKE9?usp=sharing

Verification of Code Functionality:

Searching for the word "the" in all 7 files (BUFFER_SIZE = 1024, THREAD_COUNT = 4):

```
[137] %%shell
      gcc word_count.c -o word_count
      ./word_count

      'the' is found 213 times in file /bib.
      'the' is found 507 times in file /paper1.
      'the' is found 1020 times in file /paper2.
      'the' is found 106 times in file /progc.
      'the' is found 78 times in file /progl.
      'the' is found 220 times in file /progp.
      'the' is found 156 times in file /trans.
      All files processed in 0.001166s.
```

Searching for the word “my” in all 7 files (BUFFER_SIZE = 1024, THREAD_COUNT = 1):

```
[141] %%shell
      gcc word_count.c -o word_count
      ./word_count

      'my' is found 6 times in file /bib.
      'my' is found 0 times in file /paper1.
      'my' is found 40 times in file /paper2.
      'my' is found 0 times in file /progc.
      'my' is found 1 times in file /progl.
      'my' is found 21 times in file /progp.
      'my' is found 4 times in file /trans.
      All files processed in 0.001240s.
```

Searching for the word “where” in all 7 files (BUFFER_SIZE = 1024, THREAD_COUNT = 2):

```
[149] %%shell
      gcc word_count.c -o word_count
      ./word_count

      'my' is found 6 times in file /bib.
      'my' is found 0 times in file /paper1.
      'my' is found 40 times in file /paper2.
      'my' is found 0 times in file /progc.
      'my' is found 1 times in file /progl.
      'my' is found 21 times in file /progp.
      'my' is found 4 times in file /trans.
      All files processed in 0.001045s.
```

The results seem to show a slight difference between running multithreaded processes vs running single threaded processes by just a few milliseconds, but nothing very significant on a small scale such as this. As for the accuracy of the results, I manually checked many of the files myself using a text editor and the find function (ctrl + f), and found that the number of occurrences was accurate.

Discussion of Findings:

By using multithreading, I was able to run each process and search for occurrences of words in each file marginally faster than using single threaded processes. The difference is relatively insignificant in this case, but I believe it's due to each file only being a few kilobytes in size. If I had used larger text files, maybe with a few megabytes in size, the difference might have been more noticeable. Multithreading ultimately creates more memory overhead so by reducing time you are sacrificing space, but at a scale like this running out of memory is not a concern. I also believe the negligible difference is due to colab's limitations. When running `!cat /proc/cpuinfo` on the default free tier of colab I get the specs of the CPU, showing it's a single core Intel Xeon. This leads me to believe that going above 1 or maybe 2 threads is not possible in the colab environment, and that my results would be different if I ran it on a personal machine with 4 or 8 cores with 8 or 16 threads.