

CMPSC 472: Operating Systems (Fall 2024)

**Project - 1**

File Processing System

with Multiprocessing and Multithreading

Your name: Marcos G. Ramirez

Instructor: Janghoon Yang

Submitted On: 10/23/2024.

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| 1 | INTRODUCTION | 3 |
| 2 | DESIGN OF THE DATA STRUCTURE | 3 |
| 3 | JUSTIFICATION | 3 |
| 4 | IMPLEMENTATION AND RESULTS | 4 |
| 5 | CONCLUSION | 9 |

**INTRODUCTION**

The purpose of this project is to develop a system that processes multiple large files in parallel using multiprocessing and multithreading, while employing inter-process communication mechanisms for message passing between processes. The project will allow for the comparison of multiprocessing and multithreading in terms of performance, resource consumption, and complexity.

**BACKGROUND – Multiprocessing and Multithreading**

Multiprocessing is the running of two or more programs or sequences of instructions simultaneously by a computer with more than one central processor. Multithreading is a technique by which a single set of code can be used by several processors at different stages of execution. Most computers often need to process multiple files at a given moment, and so for this project we will be using 7 files from The Calgary Corpus.

Link: <https://corpus.canterbury.ac.nz/descriptions/#calgary>

There are 14 files in this corpus, however we will only be using 7 which will be shown in the table below.

|  |  |  |  |
| --- | --- | --- | --- |
| File | Abbrev | Category | Size |
| Bib | Bib | Bibliography | 111261 |
| Paper1 | Paper1 | Technical Paper | 53161 |
| Paper2 | Paper2 | Technical Paper | 82199 |
| Progc | Progc | Source code in “C” | 39611 |
| Progl | Progl | Source code in LISP | 71646 |
| Progp | Progp | Source code in PASCAL | 49379 |
| Trans | Trans | Transcript of terminal session | 93695 |

**JUSTIFICATION**

Combining multiprocessing and multithreading leverages the strengths of both optimize resource utilization and improved performance. Multiprocessing allows a program to run multiple processes concurrently, meanwhile multithreading allows for concurrent execution within each process. By creating a hybrid approach it allows a program to maximize throughput by efficiently managing CPU resources while handling I/O operations.

**IMPLEMENTATION & RESULTS**

**The Struct and Global variables:**

In order to manage the data of the files a struct was created call ThreadData. It contains a char pointer to the name of the file that needs to be open, a wordCount int that will hold the amount of words each thread has counted. And where each thread will start and end counting within the file. This needs to be done because the thread must have an equal “snippet” of the source file. And finally, 2 global variables were defined as NUM\_FILES and NUM\_THREADS. These were created for ease of use if more threads or files were to be added later on.

#define NUM\_FILES 7

#define NUM\_THREADS 4 // 4 threads to equally divide the work

typedef struct {

  char \*file;

  int wordCount;

  long start; // Start position of the file for each thread

  long end;   // End position of the file for each thread

} ThreadData;

**Thread Process:**

The thread process needs to eventually return the number of words that were counted. So an int called NUM\_WORDS in initialized. The method for counting words uses the <ctype.h> library which will iterate through the file. But how do we count the words? By defining when a word ends. A flag variable was created called inWord that will be initialized to 0 which means we are not in a word. While scanning through the document if we come across a space or “ – “ we are still not considered in a word. Else we are in a word and we can increment the NUM\_WORDS variable. After we count through the snippet of the file we store the result in the thread data, close the file, and exit the thread. The process will also check if the file was open properly for error checking.

void \*process(void \*arg) {

  ThreadData \*data = (ThreadData \*)arg;

  int NUM\_WORDS = 0;

  int inWord = 0; // Flag to track if we are in a word or not

  // Open file

  FILE \*fileprocess;

  fileprocess = fopen(data->file, "r");

  if (fileprocess == NULL) {

    printf("FILE DID NOT OPEN\n");

  } else {

    // Move to starting point in the file

    fseek(fileprocess, data->start, SEEK\_SET);

    char reading;

    // Read until the end of the assigned snippet or until EOF

    while (ftell(fileprocess) < data->end && (reading = fgetc(fileprocess)) != EOF) {

      if (isspace(reading) || reading == '-') {

        inWord = 0;  // No longer in a word

      } else if (!inWord) {

        inWord = 1;  // New word found

        NUM\_WORDS++;

      }

    }

    fclose(fileprocess);

    data->wordCount = NUM\_WORDS;  // Store the result in the thread data

    pthread\_exit(NULL);

  }

}

**Main Function:**

The main function is where the processes will be created. First the parent process is initialized with an array of threads. Then the file names are listed in an array so that they can be easily assigned to a respective child’s process. A count is initialized that will be used for a histogram which will be gone over in more detail later. Finally a loop is created that will run equal to the NUM\_FILES defined earlier. Fork() is used to create each child process and at the start of each process a clock timer is started to measure how long each process takes. Each child process will open a file initially just to see how large each file is. Using that information and after creating the threads using the struct ThreadData created earlier, each thread will be assigned its file, the file start, file end, and a word count. The threads then begin the process described earlier in this report. When the process is done, the information is sent to the parent process with Pipes. The Parent process will organize the data and display it in a histogram. The histogram is simply an asterisk for every 1000 words to give an illusion of a graph. If the pid < 0, then the program will terminate as an error has occurred.

int main() {

  pid\_t pid;

  pthread\_t readf[NUM\_THREADS];

  // Create pipes

  int pipefd[2];

  pipe(pipefd);

  // File names that will be used in threads (CHANGE DIRECTORY LATER)

  char \*filepaths[] = {

    "/content/bib",

    "/content/paper1",

    "/content/paper2",

    "/content/progc",

    "/content/progl",

    "/content/progp",

    "/content/trans"

  };

  // Parent will store all word counts for a histogram

  int wordCountResults[NUM\_FILES] = {0};

  // Create child processes

  for (int i = 0; i < NUM\_FILES; i++) {

    pid = fork();

    if (pid == 0) {

      // Child process

      printf("Child process %d created: Process %d will open %s\n", i, i, filepaths[i]);

      // START CLOCK

      clock\_t startTime = clock();

      // Open the file to get its size

      FILE \*file = fopen(filepaths[i], "r");

      fseek(file, 0, SEEK\_END);

      long fileSize = ftell(file);  // Get the file size

      fclose(file);

      // Divide the file into equal parts

      long fileSnippet = fileSize / NUM\_THREADS;

      ThreadData data[NUM\_THREADS];

      // Create threads within child processes

      for (int j = 0; j < NUM\_THREADS; j++) {

        data[j].file = filepaths[i];

        data[j].start = j \* fileSnippet; // Start position for the reading

        // Conditional to decide the endpoint of the file snippet

        if (j == NUM\_THREADS - 1) {

          data[j].end = fileSize; // For the last thread

        } else {

          data[j].end = (j + 1) \* fileSnippet; // For other threads

        }

        data[j].wordCount = 0;

        pthread\_create(&readf[j], NULL, process, (void \*)&data[j]);

      }

      // Wait for all thread processes to finish and add results

      int totalWords = 0;

      for (int j = 0; j < NUM\_THREADS; j++) {

        pthread\_join(readf[j], NULL);

        totalWords += data[j].wordCount;

      }

      printf("Child process %d words counted: %d\n", i, totalWords);

      // STOP CLOCK

      clock\_t endTime = clock();

      double timeTaken = (double)(endTime - startTime) / CLOCKS\_PER\_SEC; // Calculate elapsed time

      printf("Child process %d completed in %.2f seconds.\n", i, timeTaken);

      // Send word count to the parent via the pipe

      close(pipefd[0]);  // Close reading end in the child

      write(pipefd[1], &totalWords, sizeof(totalWords));

      close(pipefd[1]);  // Close writing end after sending the data

      exit(0);

    } else if (pid > 0) {

      // Parent process

      printf("Parent process will wait for Child %d\n", i);

      // Parent process waits for child process

      wait(NULL);  // Wait for child processes to finish

      read(pipefd[0], &wordCountResults[i], sizeof(int));  // Read word count from the pipe

      printf("Parent has read from Child %d\n", i);

    } else {

      perror("fork failed");

    }

  }

  close(pipefd[0]);  // Close reading end of the pipe

  // Display the histogram of word counts with asterisks

  printf("\nWord Count Histogram:\n");

  for (int i = 0; i < NUM\_FILES; i++) {

    printf("File %d (%s): ", i + 1, filepaths[i]);

    // Print an asterisk for each 1000 words

    for (int j = 0; j < wordCountResults[i] / 1000; j++) {

      printf("\*");

    }

    printf(" (%d words)\n", wordCountResults[i]);

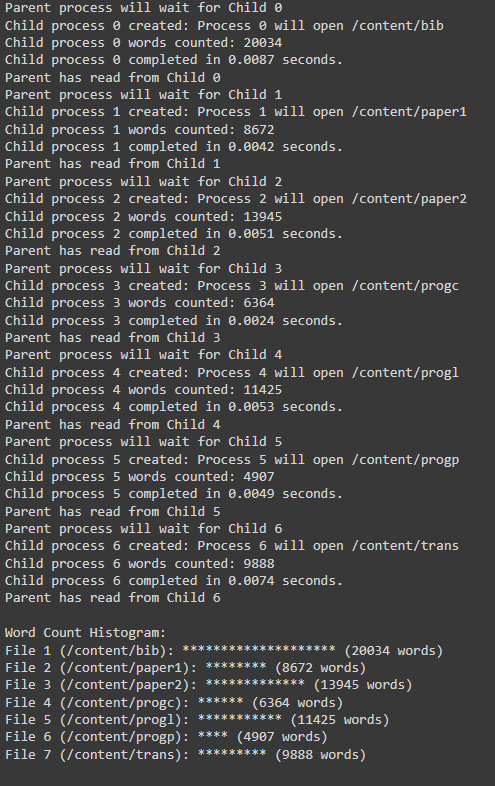
  }

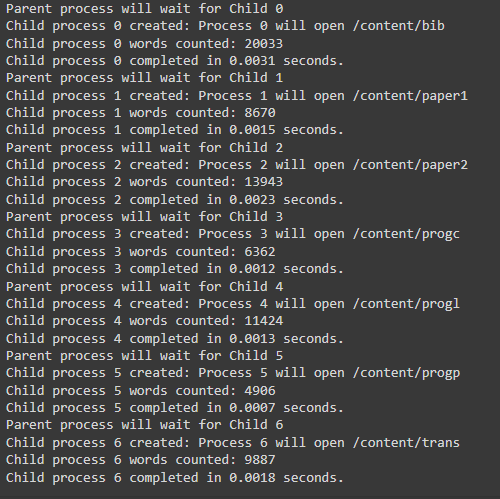
  return 0;

}

**CONCLUSION**

* **Results**:

Below this discussion will be a screenshot of the results of executing the code properly. This report goes into detail of the final code that was developed. When developing this code, I took the time to separate an instance where there was only one thread per process, and each process would have to read a file completely.



When comparing the results of just multiprocessing to multiprocessing and multithreading it would seem like just multiprocessing is faster by a noticeable margin. However, I think this is due to the file sizes not being large enough to match the time it takes for computation. Each file is being initially read to see the size of the file, then it is separated into manageable snippets. More snippets or for larger file times may be more efficient time wise.

* What have you learned from this project?

I better understand how child process and parent processes interact. When initially creating this, I was unsure of how to create enough child processes in a loop as I was concerned that child processes would be creating child processes. However, through some research I learned that it really doesn’t matter for my current project and that the first parent process will be the only “parent”. I also learned a lot more threads and structs work together for processes and can see how managing them can get complicated very quickly

* What ways can you expand this project?

Unfortunately, due to time constraints, I have not created some code verification. Namely the functionality that would count the top 50 most frequent words and their frequency. I can see how it can be done, but my attempts so far have been unsuccessful. The current code submitted will not have that functionality implemented but it is something I will work with in my free time.