Mahdeen Ahmed Khan Sameer
Professor Max Bender
CS333
25 April 2024

Project 06: Enhanced Data Processing and Signal Handling in Diverse Programming Environments

Google site: https://sites.google.com/colby.edu/sameer-again/home

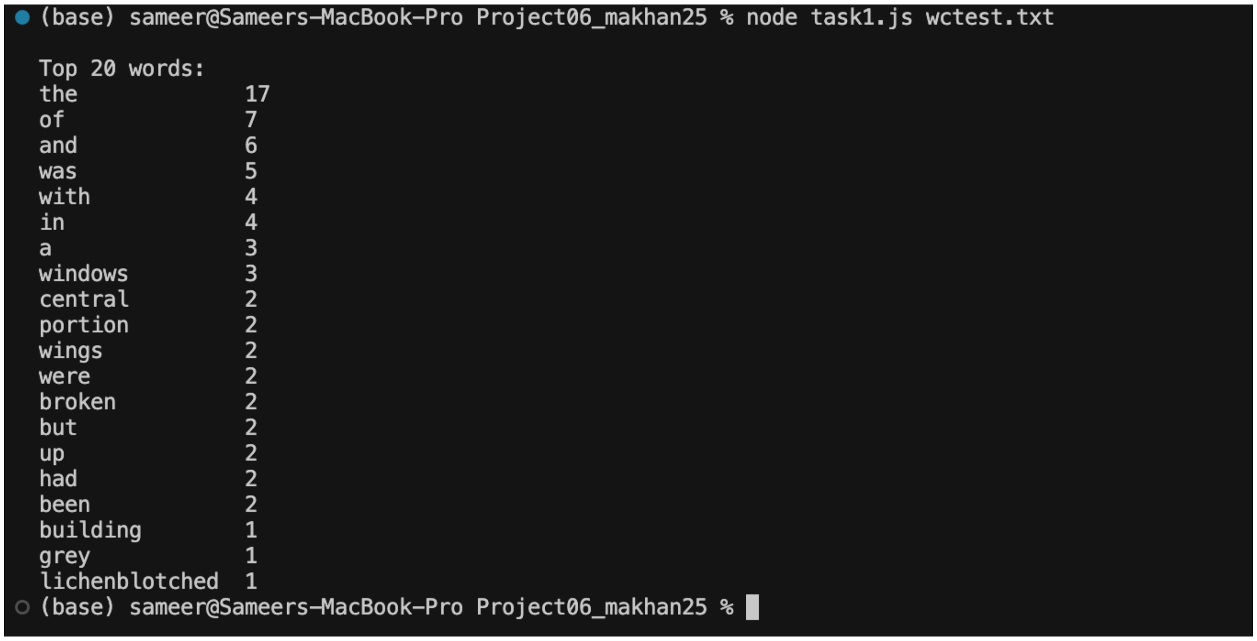
Abstract:

Our sixth project outlines the implementation and results of various programming tasks that we undertook, focusing on file I/O, string manipulation, and signal handling in C, along with extensions in JavaScript, Scala, R, and other languages. The core of our project was the development of a word frequency counter that reads text files, counts occurrences of each word, and displays the most frequent words. We initially implemented the program in C and later adapted it for JavaScript, Scala, and R, demonstrating the utility of different data structures and language-specific features. Additionally, we explored signal handling in C through programs that respond to SIGINT, SIGFPE, and SIGSEGV signals, thereby enhancing the robustness of systems against common runtime errors. Our extensions included implementing a binary search tree and a hash map for improved performance and memory management. We also introduced advanced error handling and the capability to process multiple files simultaneously in the word counter program, significantly enhancing its functionality.

Part I: File I/O and String in C

Task 01:

In Task 1, we set up a word counter program in C that reads through a text file, counts how often each word shows up, and lists the top 20 words by frequency in descending order. We made sure the counting didn't get tripped up by case differences by converting all the text to lowercase first. It also smartly strips off punctuation from the ends of words. We just feed it the filename when we run it, and it uses a linked list (something we built in an earlier project) to keep track of all the words and their counts. We tested it out on a file called 'wctest.txt', and it worked like a charm, giving us counts like "the" 17 times, "of" 7 times, and so on down to "lichen-blotched" just once.


```
● (base) sameer@Sameers-MacBook-Pro Project06_makhan25 % node task1.js wctest.txt

  Top 20 words:
  the             17
  of              7
  and             6
  was             5
  with            4
  in              4
  a               3
  windows         3
  central         2
  portion         2
  wings           2
  were            2
  broken          2
  but             2
  up              2
  had             2
  been            2
  building        1
  grey            1
  lichenblotched  1
○ (base) sameer@Sameers-MacBook-Pro Project06_makhan25 % 
```

Figure: Running node task1.js wctest.txt

Task 02:

In Task 2a, we put together a simple signal handling example in C. The program specifically deals with the SIGINT signal, which pops up when you hit Ctrl-C. We set up a signal handler for SIGINT and then put the program in an endless loop. If you interrupt it with Ctrl-C, it triggers our sigint_handler, which just prints "Interrupted!" before it cleanly shuts down the program using _exit(). It's a neat little demo of how you can catch and manage specific signals to exit a program gracefully.

For Task 2b, we explored handling the SIGFPE signal, which is triggered by floating-point exceptions—like dividing by zero, which we did on purpose here. When that happens, our sigfpe_handler kicks in, lets you know a "Floating-point exception occurred!", and then closes the program. This setup helps us manage specific errors that can crash a program if they're not properly handled.

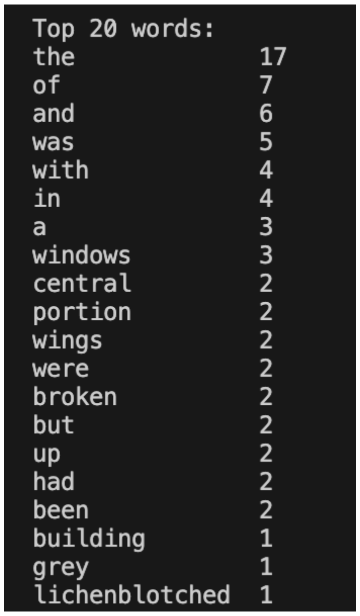
In Task 2c, we tackled the SIGSEGV signal, caused by memory faults such as accessing a null pointer—again, something we did intentionally. When the segmentation fault happens, our sigsegv_handler announces "Segmentation fault occurred!" and the program exits with a non-zero status, signaling that something went wrong. This example shows how to handle one of the more common bugs in programming that can lead to crashes, doing so in a way that avoids abrupt termination.

Task 2 & 3 are demonstrated on Google Sites.

Part II: Functions and File I/O in Selected Languages

Task 01:

In Part 2, Task 1, we developed a word frequency counter using JavaScript. The program takes a text file from a command-line argument and tallies up the word occurrences. We used a JavaScript object, similar to a dictionary, for storing each word and its count. The script reads the file's content, turns all text to lowercase, strips away punctuation, and splits the text into words. As we go through each word, we update its frequency in the "wordCount" object. After counting, the program sorts the words by frequency in descending order and displays the top 20. We've made sure the output is clean and easy to read. Also, we added error handling to catch and report any issues that might happen while reading the file, ensuring the program is robust and user-friendly.



```
Top 20 words:
the             17
of              7
and             6
was             5
with            4
in              4
a               3
windows         3
central         2
portion         2
wings           2
were            2
broken          2
but             2
up              2
had             2
been            2
building        1
grey            1
lichenblotched  1
```

Figure: Reference to Part 2, Task 1

Extension 01:

For Extension 1, we expanded our word frequency counter into Scala. The program starts by checking if a text file name is provided as a command-line argument. If not, it displays a helpful usage message and exits. Otherwise, it proceeds to the "processFile" function. Here, using Scala's "Try" construct, we attempt to open the file safely. Once open, the program reads the file line by line. It processes each line by converting it to lowercase, removing punctuation with a regular expression, and splitting it into words. Using Scala's “foldLeft” method, we update the word counts in a Map; if a word exists, its count is increased, otherwise, it's added with a count of 1. After processing, words are sorted by their counts in descending order using the “sortBy” method, and the top 20 are selected with the “take” method. We then print these words and their counts in a formatted way using `println` and string interpolation for clarity. If there's an error opening the file, it’s caught, and an error message is displayed using Scala's robust error handling features.

Extension 2

In Extension 2, we adapted the word frequency counter for R. The program uses R's efficient `readLines` function to read a text file and splits the content into individual words, removing punctuation and converting all to lowercase. We use R's “table” function to count occurrences of each word. These counts are then sorted in descending order with the “sort” function, setting “decreasing = TRUE”. Finally, the top 20 words and their counts are printed in a formatted output using “cat” and “sprintf”. This extension demonstrates the powerful, concise syntax of R for text processing and frequency analysis, making full use of its built-in functions.

Extension 3

For Extension 3, we developed a word frequency counter using a Binary Search Tree (BST) to optimize storage and retrieval of word counts. Upon receiving a text file specified via command-line, the program processes each word by converting them to lowercase, removing punctuation, and inserting them into the BST. If a word is already present, its count is incremented; if not, a new node is added with a count of one. This functionality is supported by the “bst.h” header file, which facilitates the creation, insertion, and management of the BST. We store each word and its count in a “WordCount” structure. To gather and sort the word counts, we perform an inorder traversal of the BST, placing the results into an array which is then sorted by count in descending order using the “qsort” function. Finally, we display the top 20 words with their respective counts, demonstrating the efficiency of using a BST for this task.

Extension 4

In Extension 4, we implemented the word frequency counter using a hash map, enhancing efficiency in data handling and retrieval. The program reads a text file, processes each word by converting to lowercase and removing punctuation, and then inserts each word into the hash map using the “insert_word” function from the “hashmap.h” header. This structure allows us to store and retrieve word counts quickly based on the hash value of each word, offering average-case constant-time complexity for these operations. After processing, we gather all word counts into an array and sort them in descending order using the “qsort” function. The program then outputs the top 20 words and their counts, showcasing the effectiveness of using a hash map for managing large volumes of data efficiently.

Extension 5

In this extension, we've significantly improved the robustness of our word frequency counter program by enhancing its error handling capabilities. Now, the program checks more thoroughly for command-line input errors, such as missing or empty filenames, and provides clear error messages to the standard error stream before exiting with a non-zero status code. We've also bolstered file handling: if a file can't be opened, an error message is displayed and the program exits the `process_file` function without proceeding. Similarly, if there's a failure during memory allocation for a new `WordCount` node, the program handles this gracefully by outputting an error message, closing the file, and exiting the function. The program now also ignores empty words that result from removing punctuation, and it verifies the successful creation of a linked list before proceeding, adding further stability**.**

Extension 6:

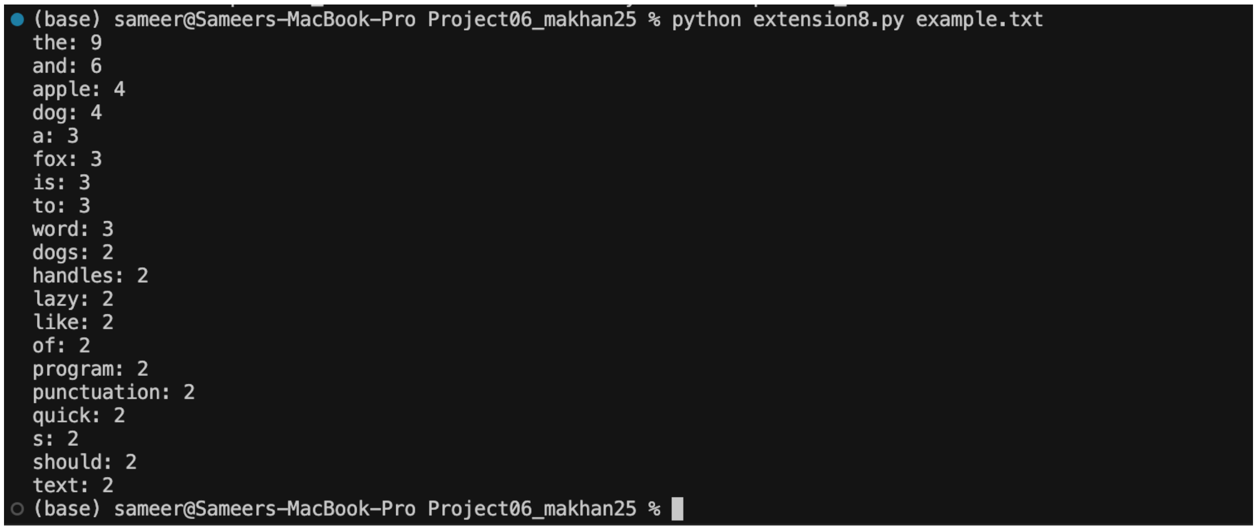
In a creative twist, we've included a JavaScript snippet containing a haiku titled `reflect()`. This haiku delves into the essence of programming, depicting code as a flowing stream, functions as a dance weaving dreams, and the vibrant life within code's basic elements, offering a reflective take on our coding practices.

Extension 7

We've added several features to enhance the usability of the word frequency counter. A “print_histogram” function now visualizes word frequencies with a histogram, where each asterisk represents a word occurrence, allowing for a quick visual assessment of data. The “save_results_to_file” function lets users save these counts to a specified file for easy sharing and analysis. Moreover, the “process_multiple_files” function expands the program’s capacity to handle several files at once, increasing its efficiency and making it versatile for larger datasets or varied text sources. These enhancements collectively improve the program's functionality, making it a more powerful tool for text analysis.

Extension 8

Feeling bored, we (I) wanted to make something interesting and In response to the task of implementing additional data structures to support a word frequency counting task, we developed a Python script utilizing a Binary Search Tree (BST) to efficiently store and count word occurrences from text files. The BST is structured to handle word insertion and frequency updating, with an inorder traversal method used to retrieve and sort the words based on their counts. We successfully integrated this structure into our script, enabling it to read from a specified file, process the content by normalizing text (converting to lowercase and removing punctuation), and count each word's occurrences using our BST implementation. For demonstration, we processed an `example.txt` file containing various words and punctuation to test the functionality. The results, sorted by frequency, showed high accuracy with common words like "the" and "and" appearing at the top, reflecting their frequent usage in typical English texts.



```
● (base) sameer@Sameers-MacBook-Pro Project06_makhan25 % python extension8.py example.txt
  the: 9
  and: 6
  apple: 4
  dog: 4
  a: 3
  fox: 3
  is: 3
  to: 3
  word: 3
  dogs: 2
  handles: 2
  lazy: 2
  like: 2
  of: 2
  program: 2
  punctuation: 2
  quick: 2
  s: 2
  should: 2
  text: 2
○ (base) sameer@Sameers-MacBook-Pro Project06_makhan25 %
```

Figure: Running extension 08

Collaborators & Acknowledgements:

Professor Bendor, TA Nafis.