Mos Kullathon
Github: mosguinz
ID: 921425216
CSC415 Operating Systems

Assignment 4 – Processing CSV Data with Threads

Description:

The purpose of this assignment is to deal with parsing CSV files, error handling, handling large datasets, summarizing data with specific criteria, utilizing vectors, implementing threads with thorough timing understanding, and preventing race conditions to ensure data integrity.

Approach:

Mostly follow the "steps" provided in the README, beginning with figuring out how to parse the CSV file.

Parsing

First, I need to figure out how to parse the CSV file. The simple approach here is to simply split on a comma. However, there are other considerations to take care of, such as fields enclosed in quotes, and newlines within fields. This means I can't simply just read the file line-by-line and split on the comma. Also, I think it would be best to avoid `strtok` completely because it mutates the original string, in addition, it would not handle blank fields e.g., "`a,,b`" because it would replace all instances of the delimiter (`,`) with `\0`.

Here's a very rough pseudocode of how I would approach parsing the CSV file.

- for each character:
    - if it's a quotation mark
        - if there is, check for another quote
            - then, this is an escaped quote
            - skip the next quote
    - if it's a comma
        - if there was an opening quote
            - then, this character is part of the field
        - otherwise
            - this is the end of the column
    - if it's a newline
        - if there was an opening quote
            - then, the character is part of the field
        - otherwise
            - this is the end of the row
    - if it's none of the above

- then, it's just a "regular" character

Processing

Looking at the screenshot in the provided README, we see that each row is separated by the call type, and for each district, we separate the response ("dispatch" and "on-scene") times into different buckets, with a special column for the aggregate.

Since the table is listed by the call type, I would store the parse data and list them by the call types. Beginning with a struct that represents it:

```
typedef struct ResponseTime
{
    ResponseType type;
    int under_2_mins;
    int mins_3_5;
    int mins_6_10;
    int over_10_mins;
} ResponseTime;

typedef struct Subfield
{
    char *name;
    ResponseTime *responseTimes[2];
} Subfield;

typedef struct CallType
{
    char *name;
    int total_count;
    Subfield *call_total;
    Subfield **subfields;
} CallType;
```

- for each row:
  - get the call type
  - get the four timestamps
  - parse and calculate deltas for dispatch and on-scene times
    - if any one of these fields do not exist, ignore the row
  - find the call type from the struct:
    - if it exists, retrieve it
      - increment the total for the call type
      - find and update the response time of the neighborhood
    - otherwise, create the call type and initialize everything
      - set the call type name
      - set the total to one

- initialize the array for neighborhood, and create them using the values passed from argv
- update the response times

Threading

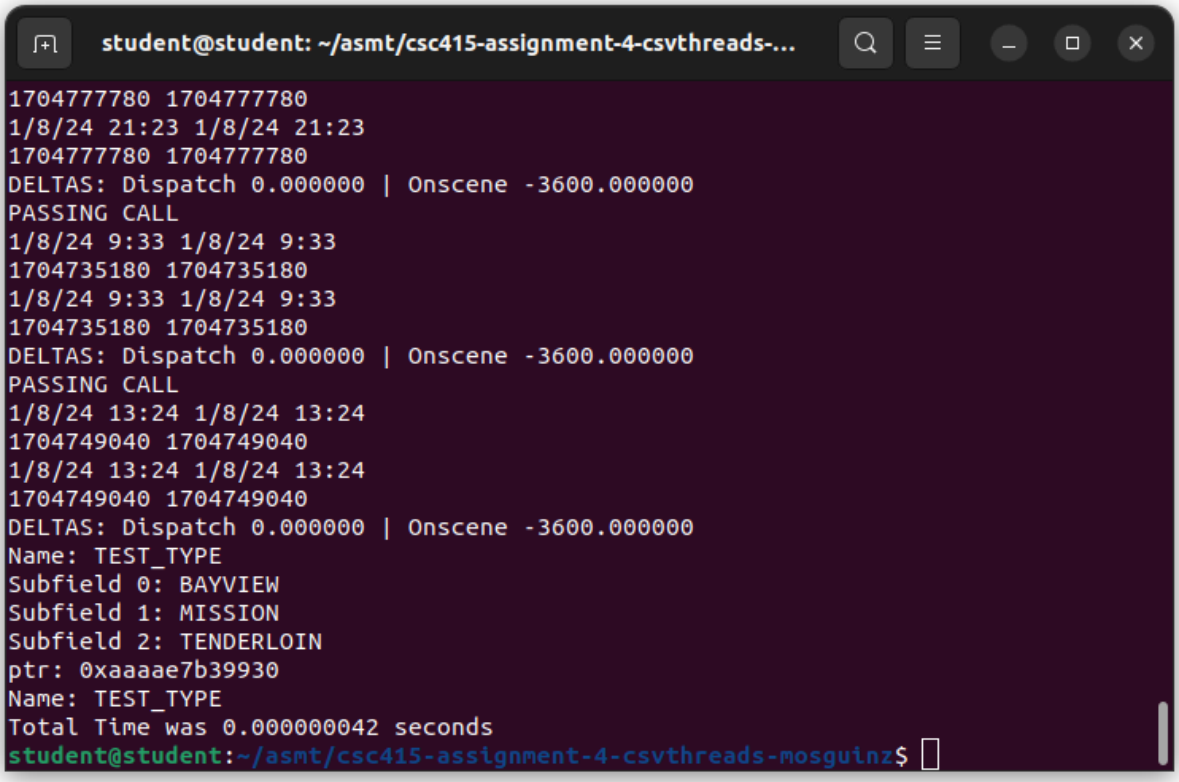
- make a function that the threads call that call csvnext
- have a mutex lock on when the file is changed
    - only on the getline, not the processing
- have another mutex lock for when the data structure is changed
    - only when values are updated, not read

Analysis:

nearly half time at 2 threads. Not fully half, because there is a mutex lock which leads to code being ran serialized. Adding more threads did not lead to a decrease in time. This is caused by the VM only having 2 cores. The threads need a core to run, so at most we can only see close to half time in the configuration of the VM. If there were 4 cores then we can expect to see close to a quarter time.

Issues and Resolutions:

Issue one: Not initializing the `tm` struct when using `strptime` to parse the time fields from the CSV. Since the provided CSV files contains two different types of timestamps, I set up the helper function to handle two formats. Not initializing the struct would cause the value return from `mktime` to be seemingly randomized. In addition, one of the format (in the 5,000-row file) do not contain seconds in the timestamp. When using `strptime` to parse, the seconds filed would be set to the current computer's time. In order to make it consistent, I needed to set the seconds field to zero, `tm.tm_sec = 0`.



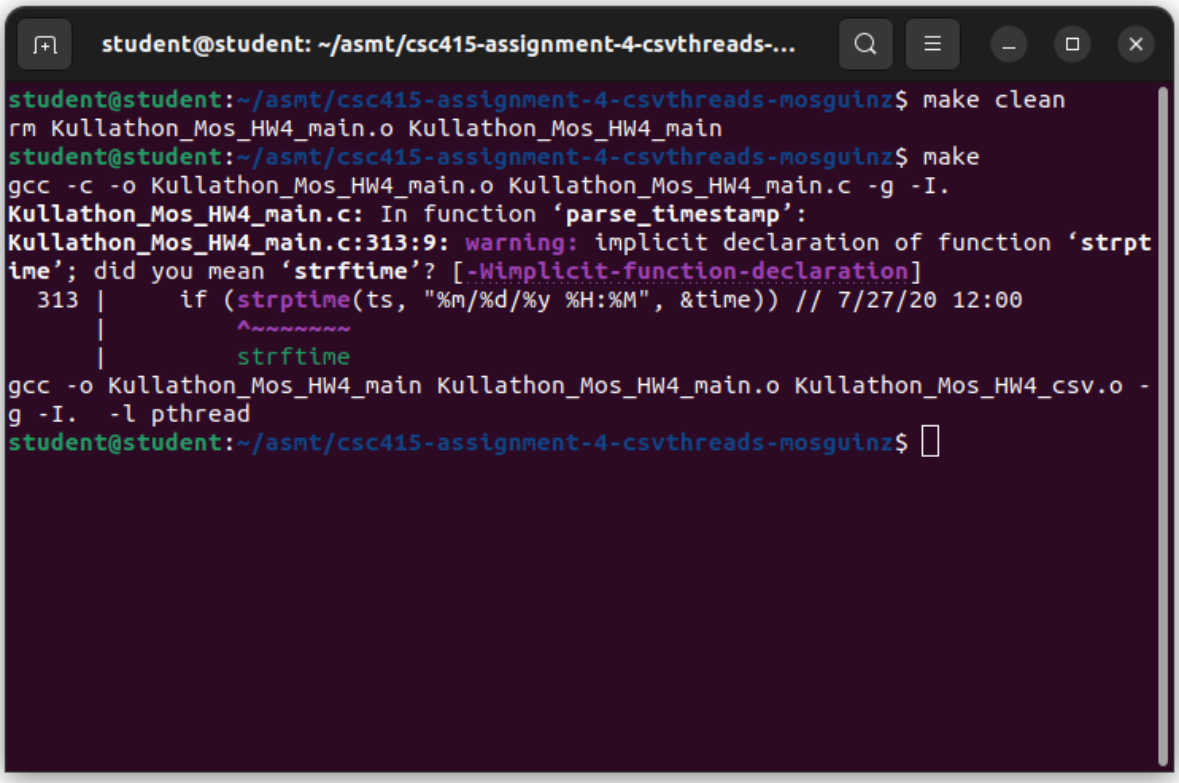
Issue two: total call count for each type being “doubled.” I had a helper function `update_calltype` to update the number of calls for each call type, for a particular response time (i.e. either on-scene or dispatch). This function is being invoked twice, once to update the call type for on-scene, and another for dispatch. I was also updating the *total* number of calls inside of this helper, which meant that they were being double counted. I fixed this by moving the incrementation of the total number of calls outside of this helper.

Issue three: Memory leak occurring when running for a certain number of rows. For the most part, I “centralize” all of my `malloc` and `free` logic into their respective functions e.g., `init_calltype` and `free_calltype` in order to properly keep track and ensure that everything was being freed at the end. However, I did not take into account where the lines being read from the CSV file were being freed at the end of the loop. When a row is determined to be “garbage,” the loop terminates early but doesn't free the row that is read.

Issue four: Structs containing string fields are producing garbage data. I initially did not heap-allocate any of the string fields. I simply assign all string values after creating the struct for each row. This did not seem to be an issue at first when simply printing in the loop for debugging purposes. However, once I fixed the memory leak above by freeing each line, the

string fields are no longer valid since they are `char*`. Instead, I had to `strdup` the all such fields, and of course update my `free_*` and `init_*` helpers.

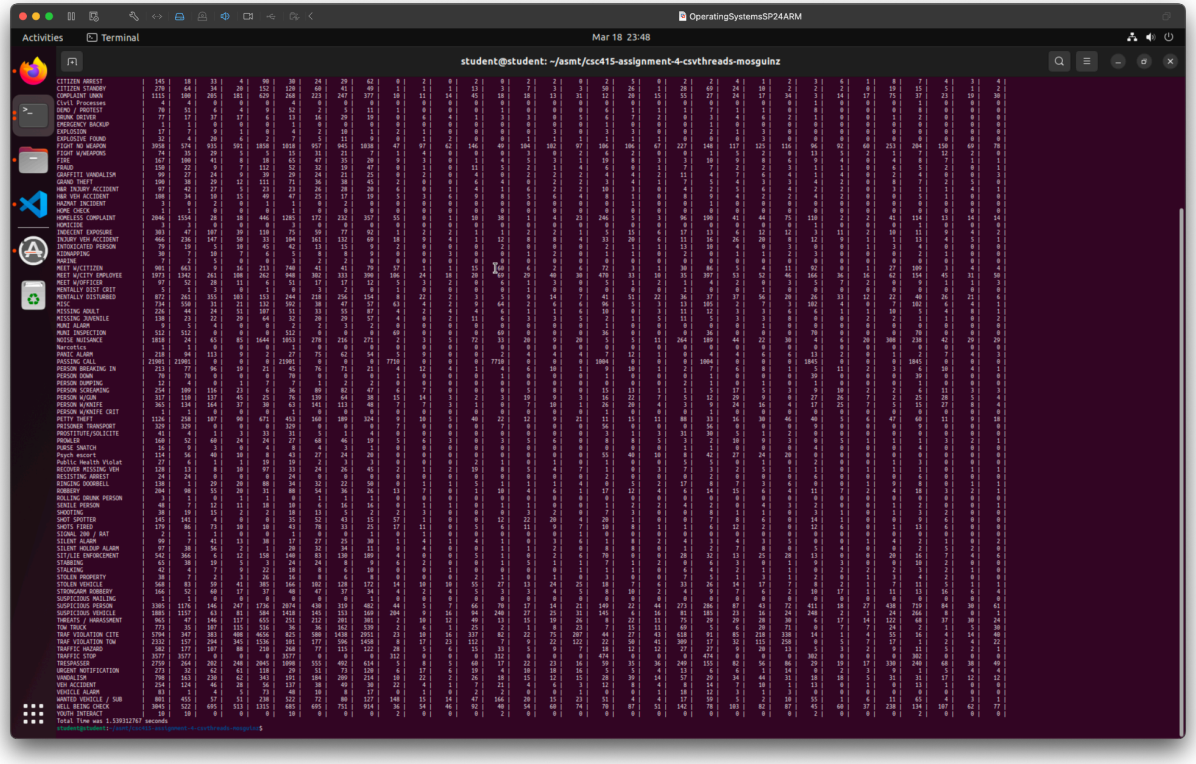
Screen shot of compilation:



Screen shot(s) of the execution of the program:

See below

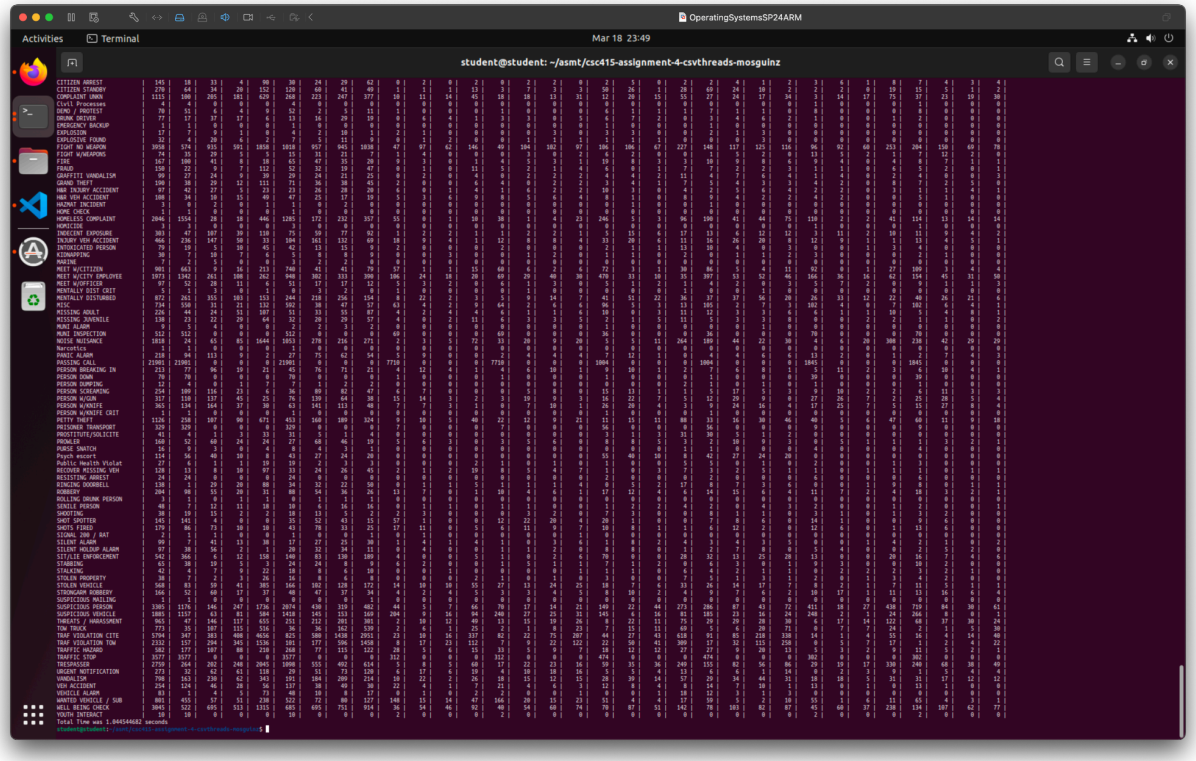
1 thread - 1.54 seconds



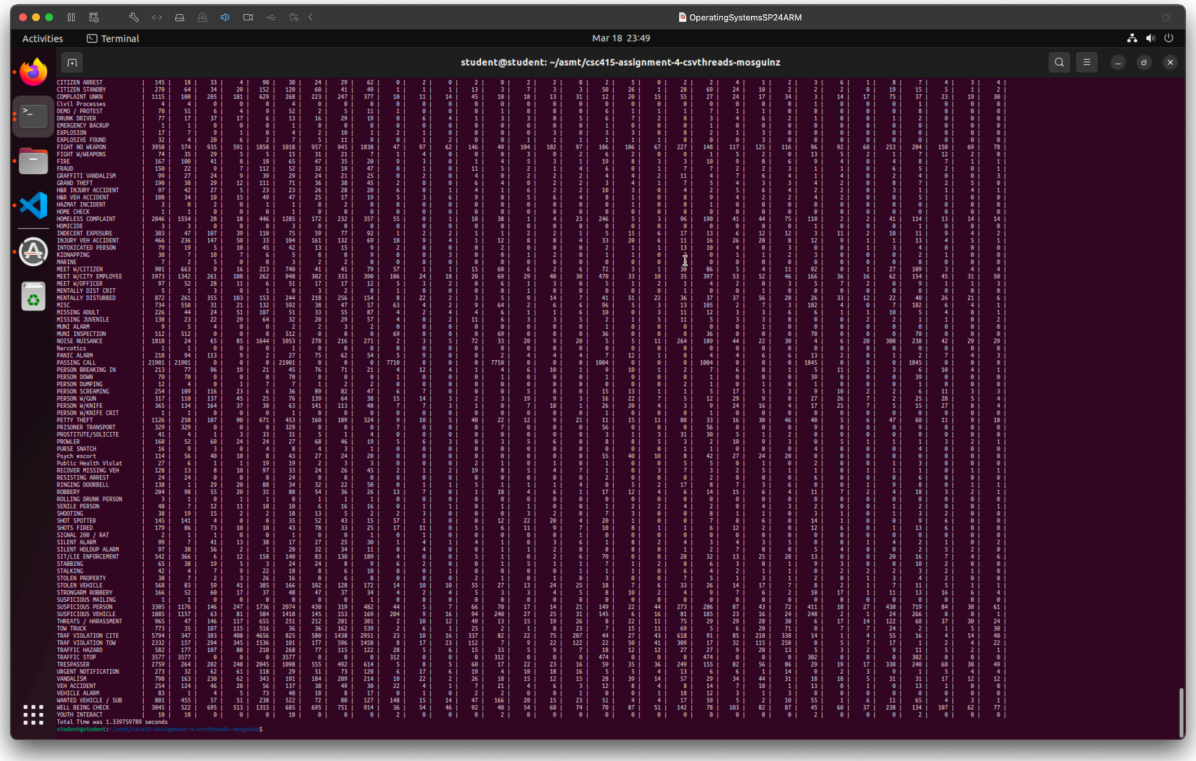
Mos Kullathon
Github: mosguinz

ID: 921425216
CSC415 Operating Systems

2 threads - 1.04 seconds



4 threads - 1.34 seconds



8 threads - 1.33 seconds