CS 4414: Recitation 1

Sagar Jha

About me

- PhD Student in CS
- Working with Prof. Ken Birman on Distributed Systems (with special focus on RDMA networks)
- TA experience at Cornell: Practicum in Database Systems (Fall 2016), Cloud Computing (Spring 2018), Cloud Computing (Spring 2020)

General Logistics

- Recitations in-person as well as live streamed on Zoom, videos will be made available for offline viewing
- Zoom users: Type questions/comments anytime in chat. Alicia (co-TA) will monitor the chat
- Email me at srj57@cornell.edu for additional questions related to the course content
- Ask questions on Piazza! Visit TAs in office hours for additional help

Today’s recitation: Emacs and C++

- A highly customizable editor for text manipulation and beyond!
- Suggested editor for the course alongside Vim
- Will be used in the recitations

- A high-level language that prioritizes performance
- Popular in systems programming
- Required for doing assignments and understanding code demos

What is a Text Editor?

- Used for writing code, latex, notes…
- E.g. Notepad, Microsoft Word, Gedit, Emacs, Vim
- IDEs: Integrated set of tools (text editor, compiler, debugger), typically for a single programming language. E.g. Eclipse, NetBeans, IntelliJ (Java), Microsoft Visual Studio (C++, C# etc.)

What is a Text Editor?

- Used for writing code, latex, notes…
- E.g. Notepad, Microsoft Word, Gedit, Emacs, Vim
- IDEs: Integrated set of tools (text editor, compiler, debugger), typically for a single programming language. E.g. Eclipse, NetBeans, IntelliJ (Java), Microsoft Visual Studio (C++, C# etc.)

What does it mean to learn an editor?

Why should one do that?

What makes Emacs special?

- Emacs has a rich set of commands for editing and movement
  - Different modes (set of shortcuts) for different types of files
- You can do almost anything in Emacs (not just editing)
  - I use git, eshell, calculator, package manager, pdf viewer
  - I have tried playing games (tetris, 2048), writing emails and browsing
- Everything in Emacs is customizable
  - From basic customizations to completely changing its behavior through emacs-lisp (you can program Emacs!)
- Emacs is self-documenting
- Emacs has many users, a large collection of third-party packages

Outline for the Emacs demo

- Working with files
- Movement within a file
- Basic and advanced editing
- Working with git
- Working with emacs-shell
- Taking notes with emacs org
- Macros
- Customizations

Resources for learning more

- Emacs built-in tutorial (C-h t)
- Emacs' internal documentation (M-x info-emacs-manual)
- https://masteringemacs.org
- https://emacswiki.org
- GNU Emacs Manual - https://gnu.org/software/emacs/manual/html_node/emacs/
- http://emacsrocks.com
- http://ergoemacs.org/emacs/emacs.html

Aside: Why use Vim?

- General rule of thumb: if you only want to focus on mostly editing one or two files in short sessions
- If you prefer separate modes for inserting text and running commands. If you find using modifiers (Ctrl, Alt) inconvenient
- Easier learning curve (probably)
- Trivia: Emacs has a vim-mode, called Evil mode!

wc++: My word count in C++

- We will not teach you C++, but I will show several code samples throughout the semester
- In this recitation:
  - Overview of basic C++ features
  - Overview of modern C++-17 features
  - Tips on efficient software management
  - Design decisions in word count for achieving high-performance
- C++ code can be written in old C-style code with global variables and functions. We strongly recommend against that. Always write object-oriented code based on proper class design.

Basic C++ features

- The RAII principle (Resource acquisition is initialization)
- Passing by reference vs. passing by value
- Separating class specification and implementation
- Using namespaces
- Standard template library (STL), specifically std::vector<T> and std::map<K, V>

Advanced C++ features

- Working with std::filesystem
- Using lambda functions
- Cheap synchronization with std::atomic<T>
- Using auto for loop
- Synchronizing std::thread(s) with std::mutex and std::unique_lock

wc++ code design

- The program takes a directory and computes word frequencies for all .h and .c files (C header and source files) in the directory
- Function main accepts user input (source dir and no. of threads)
- Main class wordCounter computes the frequencies and displays them

Designing for high performance

- Target use-case: Large no. of files with small-to-medium size (MBs)
- Important decisions:
    - How to utilize multiple threads effectively?
    - How to process a single file efficiently?
    - How to aggregate results across multiple threads quickly?

Word count with multiple threads: Real world analogy

- Suppose you have 4 people that need to process 100 books
- Processing a book is linear in the number of pages in the book
- Does it make sense to distribute the books equally among the 4 people (25 each)?

Word count with multiple threads: Real world analogy

- Suppose you have 4 people that need to process 100 books
- Processing a book is linear in the number of pages in the book
- Does it make sense to distribute the books equally among the 4 people (25 each)?
  - No! Books may have varied length. Some people might finish processing earlier than others and be idle.

Keeping each thread busy at almost all times during the execution is crucial to performance!

A dynamic assignment of files to threads

- Since I have a large number of small-to-medium sized files, each file can be fully processed by a single thread
  - Notice how the design changes if I had very few, but large sized files (GBs-TBs)
- Thus, each thread picks an unprocessed file to process. Once finished with it, it moves on to the next unprocessed file
- Synchronization issue: How to guarantee that each file is processed exactly once?
  - Performance-oriented design: Use std::atomic<int> like fetch-and-add to index into the global list of files. Saves significant time compared to using coarse-grained locking

How to manage frequency counts?

- An object of wordCounter maintains a map of a word to its frequency
- Each thread needs to update this map based on the files it processes
- Accessing this map at all times simultaneously from multiple threads is costly due to the many synchronization overheads
- Performance-oriented design: Maintain each thread’s computed frequency in a thread-local map. Update the global map at the end. Thus synchronization is only required at the end when each thread is finished processing.

How to process each file efficiently?

- Naïve solution:
  - Read file word-by-word separated by whitespace
  - Process each word to update the frequency map

    For example, **for(int** needs to be broken into two words **for** and **int**

    Similarly, **#include<iostream>** will break into **include** and **iostream**
- This is costly. It uses string functions (costly) for every word in the file separately
- Performance-oriented design: Read the entire file contents into a string. Break this entire string into tokens to update the freq-map.
  - Uses more memory to store the entire file contents, but okay since file sizes are in the range of MBs (most files in our example are a few KBs only)