210 Systems Programming
File Types, Access Rights, Globbing, Text Editors, and I/O Redirection

Fall 2025

Week 2

Overview

- File types and access rights
- Globbing
- Text editors: vim, emacs, nano.
- I/O redirection

File Types

- We will learn about the file system in detail later. For now, you should know that there are:
  - *Text files*: they can be edited with text editors and displayed with commands such as `less`. Examples: source code, configuration files.
  - *Binary files*: they have different formats and cannot be edited/displayed directly. Examples: executable programs, .docx files, .jpeg files.
  - *Directories*
  - *Symbolic links*: shortcuts to files or directories (more about them later)
- By just looking at the file name, it is not possible to distinguish files/directories, or text/binary files.
  - File extensions are for informative purposes only. There can be files with no extensions or directories with extensions. (Example: try "`mkdir a.txt`")

Multi-user Operating Systems

- Multi-user OSs like Unix/Linux should have mechanisms to protect each user's privacy and security
  - Where are each user's files located?
  - Who can access them? (What is "access"?)

The Unix Approach

- Organize *users* into *groups*
  - Each user has a unique user id and can be a member of multiple groups.
  - Use the `id` command to get information on a user:

    ```
    id tolgacan
    uid=430462(tolgacan) gid=430462(tolgacan)
    groups=430462(tolgacan),869(mioaccess),882(wendian)
    ```
- Create separate read/write/execute permissions for the *owner* (u), the *group* (g), and all the *others* (o)
  - We can use the `chmod` command to change access permissions of files and directories

Permission Syntax

- Three permission types: read, write, execute (rwx) for three classes: owner, group, others (ugo) $\Rightarrow$ 9 permission flags:
  - $r_u w_u x_u r_g w_g x_g r_o w_o x_o$
- Examples:
  - user has rwx, group has r, others have r $\Rightarrow$ 111 100 100 or `rwxr--r--`
  - user has rw, group has rw, others have none $\Rightarrow$ 110 110 000 or `rw-rw----`

Querying File Permissions

- `ls -l` will show file permissions in the first column as a 10-character string.
  - The first character shows us whether it is a directory, file, or a symbolic link.
  - `d` for directories, `-` for files, and `l` for symbolic links.
- Example: `-rwxr-xr--`
  - Owner has all permissions
  - Users in the group have read and execute permissions
  - Others have read permission

Changing permissions with `chmod`

- Can use the *symbolic* or the *numeric* mode
  - The format of a symbolic mode is `[ugoa...][[-+=][perms...]...]`, where `perms` is either zero or more letters from the set `rwxXst`, or a single letter from the set `ugo`. Multiple symbolic modes can be given, separated by commas.
  - Examples:
    - `chmod +x foo`
    - `chmod u+rwx,g+x,o-rwx foo`
  - The numeric mode use octal digits (0-7) to represent the 3-bit permissions for each ugo category.
  - Examples:
    - `chmod 754 foo`
    - `chmod 600 foo`

Execute permission on files vs on directories

- Execute permission on files means the right to execute them like commands if they are programs or shell scripts.
  - You should not give the execute permission to non-program/non-script files. If you do, the content will be treated as a sequence of commands.
- For directories, execute permission allows you to `cd` into that directory.

Practice

What are the permissions of each of the following files after each command is run? Assume initial permissions of `rwxrw-rw-`.

1. `chmod u=rwx,g=r,o= file2`
2. `chmod 441 foo`
3. `chmod u+x hello`
4. `chmod u+rwx,g-x world`
5. `chmod 643 bar`

Wildcards and Globbing

- Bash supports a system of pattern matching for referring to a set of files called *globbing*
- Although there are similarities, globbing does not use the standard Regular Expression syntax. Instead, globbing recognizes and expands wild cards, such as:.
  - * and ?,
  - character lists in square brackets,
  - and certain other special characters (such as ^ for negating the sense of a match)
- Limitations:
  - Strings containing * will not match filenames that start with a dot. Example: .bashrc.
  - The ? has a different meaning in globbing than as part of an RE.

Rules of Globbing

`*`	Matches 0 or more characters
`?`	Matches precisely one character
`[...]`	Matches specified characters in a set or in ranges
`[^...]`	Negate the set defined in square brackets
`[!...]`	Negate the set defined in square brackets
`{p1,p2,...}`	Match/expand each globbing pattern one by one

Note: In most shell implementations, ^ is used as the range negation character. However, POSIX specifies ! for this role, and therefore ! is the standard choice.

Examples

`*`	Matches any string, of any length
`foo*`	Matches any string beginning with foo
`*x*`	Matches any string containing an x (beginning, middle or end)
`*.tar.gz`	Matches any string ending with .tar.gz
`*.[ch]`	Matches any string ending with .c or .h
`foo?`	Matches foot or foo$ but not fools
`[!ab]*`	Matches any string that doesn't start with a or b
`{b*,c*,*est*}`	Matches strings that start with b, start with c, and contain est in them.

When does globbing occur?

- Bash performs filename expansion, i.e., globbing, on unquoted command-line arguments.
- In other words, in commands such as `ls`, `mv`, and `cp`, expansion is performed to convert, for example,
  - `cp *.txt dir` to
  - `cp a.txt b.txt cc.txt dir`

More on globbing

- Bash (and Korn Shell) offers extended globs, which have the expressive power of regular expressions.
- Korn shell enables these by default; in Bash, you must run the following command to enable it:
  - `shopt -s extglob`
- Additional resources on globbing:
  - Globbing reference on the Linux Documentation Project - TLDP
  - The glob page on Greg's Wiki

Text editors

- We create text content, such as README files, code, Makefiles, to-do lists, and configuration files, usually using tools with GUIs such as VS Code, notepad++.
- Discussion:
  - What are text content do you create?
  - What other tools do you use?

Command line text editors

- What if you access a remote machine via a text terminal? Would these tools will be available to you then?
- Text editors that run from within a text based terminal are useful tools to create and modify such content.
- They also offer other advantages like speed and additional functionality.
- However, their learning curve can be steeper than tools with GUIs.

Vim

- Vim is a free and open-source, screen-based text editor program.
- It is an improved clone of Bill Joy's vi.
- Vim's author, Bram Moolenaar, derived Vim from a port of the Stevie editor for Amiga and released a version to the public in 1991.

Vim

- Vim is a modal editor
  - Save keypresses by switching modes instead of using complex shortcuts
- Normal mode
  - Default mode. Used for editor commands
- Visual mode
  - Similar to normal, but used to select areas of text

Vim

- Other modes:
- Insert mode
  - Allows the user to type text into the file
- Command mode
  - Shows a single line at the bottom for normal commands (save, quit, etc.)
- You can learn Vim by using the command line program `vimtutor`

Vim Pros and Cons

- Pros:
  - Installed on most systems (even on macOS)
  - Very fast and lightweight
  - Easily perform complex edits
  - Shortcuts for everything, macros, registers, repetition, etc.
- Cons:
  - Steep learning curve

Emacs

- GNU Emacs is a free software text editor.
- It was created by GNU Project founder Richard Stallman, based on the Emacs editor developed for Unix operating systems.
- Its tag line is “the extensible self-documenting text editor.”

Emacs

- One of the oldest free software still in active development
- Uses sequences of shortcuts for operations
- Highly extensible and customizable
- You can learn emacs by launching it with `emacs` and then pressing `ctrl + h` and then `t`

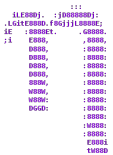
Emacs Pros

- Pros:
  - Highly extensible and customizable
  - Powerful (arguably more than any other editor, period)
  - Mature integration with other free software tools
  - You can use it for everything, even browsing the web!
  - Choose Emacs if you want a complete development environment akin to VSCode, CLion, etc.

Emacs Cons

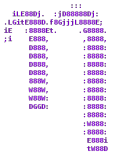
- Cons:
  - Bad ergonomics!
    - Emacs pinky syndrome
    - Recommendation: Use “Evil Mode” extension for Vim shortcuts
  - Not available everywhere. If you don’t have superuser privileges, you won’t be able to install

nano



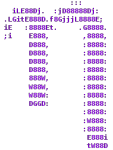
- GNU nano is a text editor for Unix-like computing systems or operating environments using a command line interface.
- It emulates the Pico text editor, part of the Pine email client, and also provides additional functionality.
- Unlike Pico, nano is licensed under the GNU General Public License.

nano



- GNU nano is keyboard-oriented, controlled with control keys.
- GNU nano puts a two-line "shortcut bar" at the bottom of the screen, listing many of the commands available in the current context.
- `ctrl + g` gets you to the help screen.

nano Pros and Cons



- Pros:
  - Easier to learn
  - GNU nano can also use pointing devices, such as a mouse, to activate functions that are on the shortcut bar, as well as position the cursor.
- Cons:
  - Not as powerful as Emacs or Vim

Which editor to use?

- "See the Editor Wars!" at:
  - https://www.linuxtrainingacademy.com/nano-emacs-vim/

I/O Streams

- There are three standard input out streams recognized by operating systems and programming languages:
  - Standard Input: `stdin`
  - Standard Output: `stdout`
  - Standard Error: `stderr`
- You should be familiar with these from CSCI 200.
  - `cin`, `cout`, `cerr`
- These streams provide basic user input and program output for C/C++, Java, Python programs and Linux commands.

Standard I/O Default Values

The default values for standard input, standard output, and standard error are:

- Standard input: The terminal/console keyboard
- Standard output: The terminal/console display screen
- Standard error: The terminal/console display screen

How do these scale for large inputs and outputs?

Example scenario: testing a program

- You want to see whether your program has any bugs by testing it on a number of test cases.

Example scenario: testing a program

- You want to see whether your program has any bugs by testing it on a number of test cases.
- How many test cases would you test it against?

Example scenario: testing a program

- You want to see whether your program has any bugs by testing it on a number of test cases.
- How many test cases would you test it against?
- 10? 100? 1000?

Example scenario: testing a program

- You want to see whether your program has any bugs by testing it on a number of test cases.
- How many test cases would you test it against?
- 10? 100? 1000?
- How long would it take?

Example scenario: testing a program

- You want to see whether your program has any bugs by testing it on a number of test cases.
- How many test cases would you test it against?
- 10? 100? 1000?
- How long would it take?
- Resolve a bug and retest?

Example scenario: testing a program

- You want to see whether your program has any bugs by testing it on a number of test cases.
- How many test cases would you test it against?
- 10? 100? 1000?
- How long would it take?
- Resolve a bug and retest?
- How would you automate the testing process?

Automating Testing

- Possible solution:
  - Read the input from files and write the output to files

Automating Testing

- Possible solution:
  - Read the input from files and write the output to files
  - Would you need to change your program to be able to process files?

Automating Testing

- Possible solution:
  - Read the input from files and write the output to files
  - Would you need to change your program to be able to process files?
  - **No need.** There is an easier way: **I/O redirection**

I/O Redirection

- You can *redirect* standard input, standard output, and standard error to files using I/O redirection

I/O Redirection

- You can *redirect* standard input, standard output, and standard error to files using I/O redirection
- `myProg < input` tells the program that the standard input is no longer the keyboard. It is the file `input`.

I/O Redirection

- You can *redirect* standard input, standard output, and standard error to files using I/O redirection
- `myProg < input` tells the program that the standard input is no longer the keyboard. It is the file `input`.
  - The characters in the file will be processed as if the user typed them with a keyboard.

I/O Redirection

- You can *redirect* standard input, standard output, and standard error to files using I/O redirection
- `myProg < input` tells the program that the standard input is no longer the keyboard. It is the file `input`.
  - The characters in the file will be processed as if the user typed them with a keyboard.
- `myProg > output` tells the program that the standard output is no longer the screen. It is the file `output`.

I/O Redirection

- You can *redirect* standard input, standard output, and standard error to files using I/O redirection
- `myProg < input` tells the program that the standard input is no longer the keyboard. It is the file `input`.
  - The characters in the file will be processed as if the user typed them with a keyboard.
- `myProg > output` tells the program that the standard output is no longer the screen. It is the file `output`.
  - The output of the program will be directly written to the file `output`.

I/O Redirection

- You can *redirect* standard input, standard output, and standard error to files using I/O redirection
- `myProg < input` tells the program that the standard input is no longer the keyboard. It is the file `input`.
  - The characters in the file will be processed as if the user typed them with a keyboard.
- `myProg > output` tells the program that the standard output is no longer the screen. It is the file `output`.
  - The output of the program will be directly written to the file `output`.
- Can do both simultaneously: `myProg < input > output`

Back to the testing problem

- How would you use I/O redirection to speed up the testing process?

Back to the testing problem

- How would you use I/O redirection to speed up the testing process?
  - Write a test generator to generate input files (Use output redirection).

Back to the testing problem

- How would you use I/O redirection to speed up the testing process?
  - Write a test generator to generate input files (Use output redirection).
  - Run your program with I/O redirection producing output files.

Back to the testing problem

- How would you use I/O redirection to speed up the testing process?
  - Write a test generator to generate input files (Use output redirection).
  - Run your program with I/O redirection producing output files.
  - Write a program to compare the generated outputs against a set of expected outputs (Can use existing Linux utilities like `diff`).

Back to the testing problem

- How would you use I/O redirection to speed up the testing process?
  - Write a test generator to generate input files (Use output redirection).
  - Run your program with I/O redirection producing output files.
  - Write a program to compare the generated outputs against a set of expected outputs (Can use existing Linux utilities like `diff`).
- The whole process can be automated with a Bash script (next week)

I/O Redirection

- You can *redirect* standard input, standard output, and standard error to files using I/O redirection
- Use < or `0<` to redirect `stdin`
- Use > or `1>` to redirect `stdout`
- Use `2>` to redirect `stderr`

I/O Redirection

Notes:

- If the output file already exists for `stdout` and `stderr` redirection, it will be overwritten
  - Use >> instead of > to *append* to the end of an existing file
- If the output file does not exist, it is created first
- If the standard input is redirected to a non-existing file, this results in an error.
- Most Linux commands are designed to be able to process input from the standard input

echo

`echo [-n | -e] string`

Displays the `string`. It is the *print* statement for the shell.

- `-n` does not print and end of line at the end.
- `-e` turns on recognition of escape sequences (useful for scripting)
  - `\b` backspace
  - `\n` newline
  - `\t` horizontal tab
  - `\v` vertical tab
  - see `man` for more

cat

`cat [-n] file ...`

Con**cat**enates each `file` and displays their content. If only one `file` is specified, just prints its contents.

- `-n` prints line numbers.
- Can use it with output redirection to merge multiple files to a single file

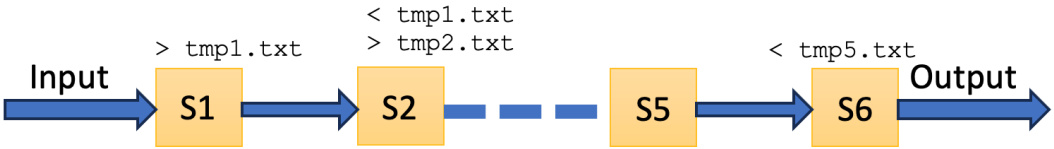
/dev/null

- If we don't care about a command's output, we can send it to `/dev/null`
- Output/files sent to /dev/null are deleted forever (recovery not possible)
  - Question: how can you send a file to `/dev/null`?
- Possible use: If we don't want error messages to clutter output, we can redirect standard error to `/dev/null`.

Using I/O redirection to connect multiple programs

- If a task requires multiple programs to work on an input in sequence, we can have them communicate with each other using I/O redirection.
- Example: spellchecking a document (See: Kernighan's Unix Demo)
  - Step 1: extract words
  - Step 2: remove punctuation
  - Step 3: convert letters to lowercase
  - Step 4: remove duplicates
  - Step 5: sort the words
  - Step 6: compare against a dictionary

Using I/O redirection to connect multiple programs



Would need a lot of temporary files in between. We will learn a better way to do this with pipes next week.