Chapter 1: Programming Tools

This handbook presents a hands-on approach to learning operating systems concepts, within the context of CSC 360 at the University of Victoria. This is intended to provide a brief overview of theory, but only within the extent needed to get started. If you, the reader, find yourself lost, you may review the suggested reading or our course textbook: "Operating System Concepts, 10th Edition".

As a preface, it should be noted that this handbook was last modified in April 2025. Since then, the technology mentioned may have evolved! However, the concepts contained within may serve as a good reference if you find yourself lost – or simply want to learn a little bit more and aren't sure where to start.

Chapter 1 covers programming tools with which some competency is expected. This chapter aims to address any gaps in understanding and establish an applied understanding of the aforementioned tools.

Authors:

Konrad Jasman

1.1. Mastering Git

Whether you're new to version control or already have a grasp of the basics, this section will help you understand how Git works, how to manage your code changes, and how to collaborate effectively with others.

Version Control is a way to track and manage changes to files or projects. It allows you to:
- Keep a history of modifications.
- Revert specific changes (or entire files) if needed.
- Work on the same project simultaneously with collaborators without overwriting each other's changes.

There are two major paradigms in version control:
1. Centralized version control (e.g., SVN), where a single server hosts the repository.
2. Distributed version control (e.g., Git), where each contributor has a complete copy (or "clone") of the repository.

Git is by far the most popular distributed version control system. It's fast, free, and powerful. The major benefits of Git include:
- Full repository copy on every contributor's machine (allowing offline work to be "pushed").
- Branching and merging are efficient and fast.
- Rich ecosystem of tools and third-party hosting services (such as GitHub, GitLab, and Bitbucket).

Anyway, by now you probably know Git is good and have a fairly strong idea of why we enjoy it so. Let's get hands-on.

1.1.1. Setting Up Git

If you are a student in CSC360 at UVic, you likely have access to the UVic CSC Linux Server – which has Git installed for your convenience. If you plan on exclusively using the CSC Servers for the duration of CSC 360, you may skip ahead to the basic configuration section.

Installing Git
- Windows: Download from git-scm.com and follow the installation wizard.
- macOS: Install using Homebrew (`brew install git`) or download the installer from git-scm.com. Brew is a fantastic package manager, and is the recommended approach.

- Linux: Install from your distribution's package manager (e.g., `sudo apt-get install git` on Debian/Ubuntu).

Basic Configuration

After installing Git, it's essential to configure your username and email address. These details are stored with every commit and encourage accountability per each user – an important detail when working online.

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

You can also customize Git's behavior by editing the global `~/.gitconfig` file. For example:

```
git config --global core.editor "nano"
```

This sets your default text editor for commit messages to Nano. You can also set it to Vim, VS Code, or any other editor you prefer. If you are just starting out, it will likely be most convenient to instead set this to your favorite editor.

1.1.2. Creating a Repository

A repository (often abbreviated to "repo") is a collection of files and the history of changes made to those files. There are two ways to create a new repository:

1. Initialize a local folder:

```
mkdir my_project
cd my_project
git init
```

- This command sets up a new Git repository in the current directory, creating a hidden `.git` folder that stores all version information.

2. Clone an existing repository:

```
git clone https://github.com/username/example-repo.git
```

- This downloads the entire repository and its history to your local machine.

If you choose option 2 and wish to use a remote repository, you will need to add a remote "origin" – the place where your files will be backed up (see: Adding a Remote Origin in the Remote Operations section).

Git works in three main areas:
1. Working Directory: Your local files/folders where you make changes.

2. Staging Area (Index): Where you place changes that you plan to include in the next commit. You add content here with `git add`. Typically, files in the staging area will be committed to your repository with `git commit`.
3. Repository: The place where commits (snapshots of your changes) are stored permanently. Your repository is primarily local, but may be mirrored onto a service such as Gitlab or Github.

Here's the standard cycle:
1. Edit files in your working directory.
2. Stage changes using `git add .`
3. Commit changes to record a snapshot in the Git history.

1.2. Basic Git Operations

Checking the Status

```
git status
```

- Tells you which files have changed, which are staged, and which are unstaged.
- Essential for understanding your current state before committing.

Adding Changes to the Staging Area

```
git add <file_or_folder>
```

- Moves changes from the working directory to the staging area.
- Common patterns:
  - `git add file.txt`
  - `git add .` to add all changed files in the current directory (and subdirectories).

Committing Changes

```
git commit -m "Commit message"
```

- Creates a snapshot of all staged changes.

Commit messages should be descriptive enough to explain why the changes were made. A good commit message example might be: "Fix user authentication timeout issue". Often, commits are done sparingly: committing only when there are significant changes to push. However, you may find yourself committing significantly more throughout the duration of the course. This is fine: you are learning, after all!

One important reason for commit messages is for you, the developer. Say you want to rollback to a previous version of your code. You could look through each previous commit and your inspect changes, but this is quite time consuming. If you use informative commit messages, you can quickly rollback to a certain state of your program.

Viewing Commit History

So… you are working on a longer assignment with multiple parts. After many hours of work, you get one part working. Great! Time to tackle the next.

A few hours in to the next part, you suddenly realize that your approach is entirely wrong. Maybe in the process you modified the previously working code and now your entire program doesn't work. Shoot! You're tired and find yourself in error hell: unable to revert your program to it's previously working state. Do you need to restart from the beginning and toss out all your hard work? No! Because you are an astute developer that used Git for their project and pushed their work after implementing your solution to the first part, you can use `git log` to view a history of each change, and prepare to rollback to a previous version.

```
git log
```

- Shows a list of commits, including commit IDs, authors, dates, and commit messages.
- Press q to quit the log view.

Short variants:

```
git log --oneline
```

- A compact, single-line summary of each commit.

1.2.1. Version Control Operations

Reverting and Resetting Commits

In the previous example, we learned how to view a history of all your commits. So, now we have two seemingly similar and yet very different commands: `revert` and `reset`.

`Revert` functions similarly to "undo": if your commit introduces a bug, you can revert it with a new "undo" commit.

```
git revert <commit-hash>
```

- This creates a new commit that negates the changes made in `<commit-hash>`.

`Reset` is a bit more aggressive: moving the current branch to a specific commit and discarding or keeping certain changes depending on the reset type. This is a good option if you simply want to make the code in a certain branch, likely the one you are currently working on, have the same content as a previous commit. `Reset` may be configured with several flags:

1. Soft reset (`--soft`): Keeps your changes in the staging area.

2. Mixed reset (`--mixed`): Keeps your changes in your working directory, but unstaged.
3. Hard reset (`--hard`): Discards all changes in your working directory and staging area.

Remember that *working directory* is the current space you are working in on your device. *Staging area* is the space between your working directory and your (local) git repository.

For example:

```
git reset --hard <commit-hash>
```

- This permanently removes commits that come after `<commit-hash>` from your local branch.

Pulling and Pushing Changes

The pull and push commands form the backbone of Git's collaboration.

`git push` sends your committed changes from your local repository to a remote repository, making them available to others. This is how you share your work, backup your progress, and contribute to shared projects.

`git pull` does the opposite—it fetches the latest commits from a remote repository and integrates them into your local branch, keeping your work synchronized with the team's changes.

These commands bridge your isolated development environment with the broader collaborative workspace!

```
git push <remote> <branch>
```

- Uploads your local commits to a remote repository, commonly on GitHub, GitLab, or Bitbucket.
- Example: `git push origin main`.

```
git pull <remote> <branch>
```

- Fetches the latest changes from the remote repo and merges them into your local branch.
- Example: `git pull origin main`.

A very similar command to `git pull` is `git fetch`. `fetch` downloads changes from a remote repository and emplaces them in your local repository, but unlike `pull`, does not integrate them into your working files. This is somewhat safer, as it allows you to inspect remote changes before integrating them. However, as with most things, is mildly more inconvenient to perform.

1.2.2. Branch Management

Working with Branches

Branches in Git allow you to create independent lines of development within the same repository. One of the more common uses of branching is in Trunk-based development: where one "stable" branch is maintained that is well-maintained and guaranteed to work.

Branching is crucial for experimenting with new features, bug fixes, or alternative ideas without disturbing the main codebase.

Creating and Switching Branches:

```
# Create a new branch
git branch feature/new-feature

# Switch to the new branch
git checkout feature/new-feature

# Combined shortcut: create and switch in one command
git checkout -b feature/new-feature
```

Merging Branches:

When your feature is complete, or you wish to move your branch's changes back to the main branch, you may merge the changes back into another branch (often "main" or "dev"). There are a variety of ways to do this, however the safer approach is first merging the destination branch into your current branch, then merging the result back to your desired destination: this means that any mistakes made during merge resolution will not impact the "good" destination branch.

Safer approach:

1. From feature branch:

```
git merge main
```

2. Resolve any merge conflicts if necessary, then commit the merge.
3. Switch to the branch that you want to merge into:

```
git checkout main
```

4. Merge your feature branch:

```
git merge <feature-branch-name>
```

Deleting a Branch

The rules for branch deletion vary per organization. Generally, branches are deleted after merging or if the branch is no longer needed:

```
git branch -d feature/new-feature
```

- The `-d` option deletes the branch if it has been merged.
- Use `-D` (capital D) to force-delete a branch that may have unmerged changes.

Handling Merge Conflicts

A merge conflict occurs when two different changes clash in the same file area. Git can't automatically decide which change to keep. Instead, it marks the file with conflict markers and lets you decide.

For example, you might see:

```
<<<<<<< HEAD
this is some text from branch A
=======
this is some text from branch B
>>>>>>> feature/new-feature
```

How to resolve a merge conflict:
1. Open the file containing the conflict.
2. Look for the conflict markers (<<<<<<<, =======, >>>>>>>).
3. Decide how to combine or choose changes.
4. Delete the conflict markers. Some editors handle this automatically depending on your selection; others do not.
5. Save the file.
6. Stage and commit the resolved file:

```
git add conflicted-file.txt
git commit
```

This can be quite tricky, and you will likely make mistakes – especially the first few times. Be careful!

1.2.3. Remote Operations

Working with Remotes

A remote is a copy of your repository stored elsewhere (usually on a server or hosted platform). Typically:

- `origin` is the default name for your primary remote.

Listing Remotes

```
git remote -v
```

- Shows the short name (e.g., origin) and the corresponding URL.

Adding a Remote Origin

```
git remote add myremote https://github.com/username/myrepo.git
```

- `myremote` becomes your short name for that remote URL.

Removing a Remote

```
git remote remove myremote
```

1.2.4. Advanced Git Techniques

Stashing Changes

Stashing allows you to save your uncommitted changes temporarily without committing them. You might want to stash your changes temporarily to allow for pulling changes from a remote source.

```
git stash
```

- Removes changes from your working directory but saves them on a stack.

When you're ready to bring them back:

```
git stash pop
```

- Restores the most recent stash and removes it from the stack.

Or view the list of stashes:

```
git stash list
```

- And apply a specific stash:

```
git stash apply stash@{2}
```

If you no longer want a stash:

```
git stash drop
```

- Removes the most recent stash.

Rebasing

Rebasing allows you to change the base of your branch to another commit, creating a linear project history.

```
git checkout feature/new-feature
git rebase main
```

- Applies the commits of feature/new-feature on top of the latest main branch.

Interactive rebase (`git rebase -i <commit_or_branch>`) allows you to reorder, squash, or edit commits. This is more advanced but highly useful for cleaning up commit history.

1.3. Best Practices and Tips

1. Commit Often and Write Descriptive Messages
    - Frequent commits help isolate changes.
    - Good commit messages explain why you made changes, not just what you did.
2. Use Branches for Features and Bug Fixes
    - Work on new features or fixes in separate branches.
    - Avoid pushing incomplete work directly to main.
3. Pull Before You Push
    - Regularly pull the latest changes from your remote to reduce merge conflicts.
4. Keep Your History Clean
    - Consider using rebase (instead of merge) for feature branches if you want a linear history.
    - Use squashing to reduce multiple small commits into one logical commit.
5. Be Mindful of `git reset --hard`
    - It can permanently delete changes. Use it with caution.
6. Leverage `.gitignore` Files
    - Ensure large files, temporary files, or system files don't clutter your repo.
    - Example entries in .gitignore:

```
# macOS
.DS_Store

# Windows
Thumbs.db

# Python
__pycache__/
*.pyc
```

```
# Node.js
node_modules/
```

There are many more advanced Git concept, from Cherry-Picking to Submodules and more. If you are interested, the best source of information is the Git book itself, found here.

Linux Command Line Basics

Similar to Git, it is difficult to progress through your career as a software developer without running into Linux. For many who primarily use Windows or Mac systems, you can get around your system fairly well without using Bash/Zsh, Powershell, or alternative command line client. Not so for Linux. Most Linux systems are faster and more effective to navigate via command line.

Makefile Basics

Makefiles allow us to simplify compilation. Generally, compiling a series of files correctly can be complicated; requiring fairly lengthy strings.

When you're working on software projects, especially those with multiple source files, the compilation process can quickly become tedious and error-prone. Let me illustrate this with a motivating example. Imagine you're developing a simple C program with the following structure:

- `main.c` - Contains your main function
- `utils.c` - Contains utility functions
- `math_operations.c` - Contains various math operations
- Plus additional header files for each source file as needed

Without a makefile, you'd need to type out the following manually in order to compile your program:

```
gcc -c main.c -o main.o
gcc -c utils.c -o utils.o
gcc -c math_operations.c -o math_operations.o
gcc main.o utils.o math_operations.o -o my_program
```

This might seem manageable initially, but consider how this combination of commands gets worse as we work with more files. The solution? Makefiles. Makefiles are single files, most often titled simply `makefile`.

Here is an example `makefile` that performs the same task, when run with `make` or `make my_program`, will produce an executable titled `my_program`:

```
CFLAGS = -Wall -Werror

myprogram: main.o utils.o math_operations.o
  gcc main.o utils.o math_operations.o -o myprogram

main.o: main.c
  gcc $(CFLAGS) -c main.c -o main.o

utils.o: utils.c
  gcc $(CFLAGS) -c utils.c -o utils.o

math_operations.o: math_operations.c
  gcc $(CFLAGS) -c math_operations.c -o math_operations.o

clean:
  rm -f main.o utils.o math_operations.o myprogram
```

Makefiles have several components. Let's break down the example above:

- The `CFLAGS` variable is conventionally used to set compilation flags, typically used to aid in development and debugging. `-Wall` and `-Wpedantic` are the most notable tags, however mayn more exist. For more compiler flags, see here.
- Rules: each rule is structured in the format `target: dependencies` and is followed by a command. We use rules to inform `make` what needs to built, and how it should be built.
- Targets: Targets are files or actions that make will create or execute; they appear on the left side of the colon in a rule. In the above example, a target might be `mypgrogram`, which would create an executable `myprogram`, by utilizing a series of sub-targets `utils.o` and `main.o`. To run this target, we make execute `make myprogram` in our bash/zsh client.

We may also have `phony` targets which represent not an actual file to be created, but instead an action to be performed. `Clean` is the best example of this, as it is often included to clean up the files produced by make. Here is an example:

```
clean:
   rm -f *.o
```

Note that the default target of `make` is the first target in the makefile. This is the target that will be run when you run `make` without specifying a target.

- Dependencies are the files that need to exist and be up-to-date before the target can be built. In the above example `main.o`, `utils.o` and `math_operations.o` are dependencies. Make checks the timestamps of a target and all its dependencies, and rebuilds the target if any dependency is newer than the target. Often times a dependency will have its own run in the makefile. This is called a `dependency chain`.

Makefiles can be used with automatic variables to reduce the verbosity. Here are the more common automatic variables:

- `$@` : refers to the target of the current rule.
- `$<` : refers to the first dependency – otherwise known as the first filename after the colon in your rule.
- `$^` : refers to all dependencies, separated by spaces.
- `$?` : refers to all variables newer than the target.
- `$*` : refers to text that matches the `%` wildcard in a pattern rule. It gives you just the matching part, without any extensions. For example:

```
%.o: %.c
  gcc -c $< -o $@
  echo "Compiled $* module" >> build.log
```

Here, the wildcard `%` matches `main`, so the command should execute equivalent to running the following from your shell terminal:

```
    gcc -c main.c -o main.o
    echo "Compiled main module" >> build.log
```

If we were to streamline the complete example above using automatic variables, we might get something like the following:

```
CC = gcc
CFLAGS = -Wall -Werror
OBJECTS = main.o utils.o math_operations.o

myprogram: $(OBJECTS)
  $(CC) $^ -o $@

main.o: main.c
  $(CC) $(CFLAGS) -c $< -o $@

utils.o: utils.c
  $(CC) $(CFLAGS) -c $< -o $@

math_operations.o: math_operations.c
  $(CC) $(CFLAGS) -c $< -o $@
```

```
clean:
  rm -f $(OBJECTS) myprogram
```

For the complete list of automatic variable, see the makefile documentation.

If you are familiar with Linux commands, you might notice here that we use `echo` within a makefile and wonder: “Hmm, I thought makefiles were only for building C programs.” Nope! Makefiles have much more extensible uses. You can use makefiles as a way to automate more advanced build processes in many languages from Java to Python, manage data pipelines, handle deployment tasks, run tests, as well as compile LaTeX and Typst documents like this one more efficiently. However, just because you can use makefiles to do it, does not necessarily mean they are the best tool for the job. Keep this in mind in your career, and enjoy the beauty of makefiles!