Midterm Exam

Name: Alex Karwowski

Email: akk42@zips.uakron.edu

Abstract

There are many tools used everyday in the development process when it comes to programming. Some of these tools are older(`bash`, `git`, and `cmake`), while others are fairly new only coming around into the limelight recently (`docker`, `ninja`, and `systemd`). Regardless of how old or new these tools are, they are used extensively and are extremely important. In this paper, I will be explaining the usage of many of theses tools and what part of the development process they are most commonly used.

Coding

Git

Using `git` is essential for any professional developer. `git` is one of the most popular and most common tools used in the development environment today. It was created by Linus Torvalds to combat old outdated Concurrent Versions System (CVS).

`git` Is a form of Version Control (VC). Version control is a system of tools that are used to manage computer programs, documents, websites, and other types of constantly changing information and files. `git` is useful in that when you use `git` you create a repository that tracks any and all changes to files, folders, programs, and more. This allows you, as a programmer, project lead, or head developer to follow and track changes made by you and others to your source code and allow for administration of new changes via pull requests, branches, and commits.

Using `git`

`git` has a number of features, so many that a single paper wouldn't be able to cover them all. Instead, here are the core features that every developer should be familiar with.

- `git add` - this command adds a file(s) to a staging area where you can make sure everything is set before committing to the repository. Any file not added via this command will not show up in the repository as changed.
- `git commit` - often followed by the `-m` flag, this command takes all files included in the `git add` command and actually "publishes" the changes to your local repository. This does not make any changes to the remote repository that everyone else has access to. Usually commits are followed by a message, (the `-m` flag) that states "By applying this commit I will..." and then the actual commit message.
- `git push` - this command takes all local repository commits, and "pushes" them to the remote repository. After pushing your commits, other collaborators can see the changes you made as well as the messages attached to those changes.
- `git pull` - when you are collaborating with others on a git repository, you should perform this command as often as possible. This command fetches and applies any changes collaborators have pushed to the remote repository.

Below is an example of the git commands I used to push changes for this markdown to the online repository

```
$ git add Report.md
$ git commit -m "Add git command examples in the Using git section"
$ #Once I am ready to push changes
$ git push
$ #After not touching the code for a couple hours its always a safe idea to pull
$ git pull
```

GitHub

Github is an online repository that allows people to host their repository's for free (for personal use). They also have private repositories, as well as Team and Business plans for group and business work. GitHub is a very popular online repository hosting services and therefore is something that any developer should be aware of. Storing your repositories online makes it easy for you to access your source code on multiple machines anywhere you have internet access.

GitHub Alternative: Bitbucket

Another Alternative to GitHub is a site called BitBucket. BitBucket seems to be more business oriented rather than open-source oriented so a lot of businesses with private code bases use BitBucket instead of GitHub.

Debugging

Unix Shells

Most programs are run on a CLI or Command Line Interface, and usually run on unix distributions rather than Windows. Because a lot of development is done with CLI in mind a lot of debugging tools are used via these unix shells. Some of the most popular shells are `bash`, `zsh`, and `fish` and they all have a variety of system functions, as well as aliases, custom written shell functions, and scripts to help in the debugging process.

To see an example of how to write Shell Aliases or Shell functions read the Testing Section below.

Note: Aliases and Shell functions are only temporary unless they are added to the `~/.{shell}rc` or `~/.{shell}_profile` files.

With all these abilities baked into most modern terminals, repeating long commands and tasks becomes a breeze and helps speed up debugging time since you no longer have to worry about erroneous commands and you can focus on the actual debugging part.

Building

CMake

`CMake` is a cross-platform build tool designed to build, test, and package software. `CMake` allows developers to make a simple unified configuration or make file to allow anyone to build, test, and deploy source code. Generally this is done by creating a `CMake` file named CMakeLists.txt that contains configurations and commands to generate a `make` file (or other file depending on the generator used) that can then be used to run a multitude of tasks.

To generate build files like Makefile or other generator files perform the following command. You can try this out using this repository.

```
$ cmake .
$ #Default generator is used
$ cmake . -G Make
```

```
$ #<Generator> is any of the supported generators used by cmake such as Make,
Ninja, or XCode.
$ cmake . -DLINK_STATIC=OFF
$ #The -D flag can be used to change the default options set inside the
CMakeLists.txt file.
$ #The option enabled to off makes the program compile with a shared library
rather than a static library
```

Make

`Make` is a tool that runs a Makefile, which is a set of instructions that are needed to perform an action (usually compiling a program or running a program). You can write a simple Makefile if you need a simple compile, run, and clean script to run. But more commonly complex Makefiles are generated by build tools like `CMake`. To generate a Makefile via cmake perform the following commands in your terminal.

```
$ cmake .
$ #Make is the default cmake generator.
$ #If for some reason it is not the default do the following
$ cmake . -G Make
```

To run your Makefile perform the following commands.

```
$ make
$ #This builds the "all:" command or other default command defined in your
Makefile.
$ #Some common commands to use with make are
$ make run
$ #runs the program
$ make test
$ #this is the command usually used for running test files or unit tests on a
program.
$ make install
$ #installs the program based on the CMakeLists.txt configuration
$ make clean
$ #deletes build files like .o or .obj and executables generated by compilation
and running the file
```

Make Alternative: Ninja

There are multiple alternative make systems besides make. A popular and fast alternative is called Ninja or Ninja-Build. This make system is decently fast and optimized. To generate a Ninja build file, when you perform your cmake command do the following

```
$ cmake . -G Ninja
$ #The "-G Ninja" Tells Cmake what type of file to generate.
```

To run your Ninja build file perform the following commands.

```
$ ninja
$ #This builds and compiles your program by default
$ #Some common commands to use with ninja are:
$ ninja run
$ #runs the program
$ ninja install
$ #installs the program based on the CMakeLists.txt configuration
$ ninja test
$ #this is the command usually used for running test files or unit tests on a
program.
$ ninja clean
$ #deletes build files like .o or .obj and executables generated by compilation
and running the file
```

Testing

Docker

Docker is a super useful tool for managing testing environments called containers. Each container acts like its own self-contained environment that has the minimal size and functionality needed to run and test programs consistently. With the ability to fine tune what packages and dependencies are installed in an environment, using docker containers is crucial for smooth testing.

Normally Docker only lets you make containers based off of the OS you are running it on (Windows only works with Windows Containers, and Unix with other Unix containers). But due to advancements in Windows Subsystem for Linux (WSL), Windows 10 users can effectively create a container for any environment using WSL as a sort of backend that handles the system calls.

Like GitHub mentioned earlier, Docker has DockerHub which allows you to "pull" different containers off a online repository. These images could be official images created by the OS's developers like Ubuntu, Fedora or even Arch Linux, as well as other non OS related containers such as Database servers like couchbase, MongoDB, and more.

Another way to create containers is to make a `Dockerfile`. In the Examples folder in this directory there is a premade Dockerfile for this project that you can use to mess around with. Below I will go over the basics of creating a Dockerfile.

```
# The FROM states what image on DockerHub to pull from to build a base container
off of.
FROM ubuntu:20.10

# LABEL states the details for this container, like the distro its being built off
of, as well as the maintainers name, email, and any other relevant information
LABEL com.3cdigitalmedianetwork.distro="ubuntu"
```

```
# RUN tell the container what commands to run when building the container
originally. This is where you would type what kind of packages you want to
install, any updates you need to do, as well as any other commands that need to
automatically be run before a user starts a container (auto run programs, setup a
server, set Environment variables, etc)
RUN apt-get update && apt-get install -y \
    # Update the repository and install the following packages
    # Install git for repository purposes
    git \
    # Install Curl for our next Run command
    curl \
    # Install make for building
    make \
    # Install g++ for compiling
    g++ \
    # Finally clean and remove any lists to cut down on image size
    && rm -rf /var/lib/apt/lists/*

# ARG let you define a variable for the Dockerfile that can be used to easily make
changes without having to fix every line.
# Install cmake binary directly
ARG CMAKE_VERSION=3.18.4
# To call an ARG generally you use the form $ARG_NAME or ${ARG_NAME}
ARG CMAKE_URL=https://cmake.org/files/v3.18/cmake-${CMAKE_VERSION}-Linux-
x86_64.tar.gz
# Here we run curl to install cmake from source.
RUN curl -L $CMAKE_URL | tar xvz --strip-components=1 -C /usr/local

# ADD is used to add a file into a chosen directory
ARG CLI11_VERSION=1.9.1
ARG
CLI11_URL=https://github.com/CLIUtils/CLI11/releases/download/v${CLI11_VERSION}/CL
I11.hpp
ADD $CLI11_URL /usr/local/include/
```

Once you have a docker file created, run the following command to build it.

Note: on Windows 10 with WSL, the Docker sub-service needs to be started in order to use any of the docker commands

```
$ docker build . -t testBuild
$ #This will create a container called testBuild based on the Dockerfile located
in the current directory (.)
$ #To change the location for the Dockerfile change "." to "Path/to/Dockerfile"
$ #Now you can run the following command to start the container.
$ docker run -it testBuild
$ #The -it allows you to run the container interactively
```

Docker Alternative: Virtual Machines

An alternative to using docker containers is to use virtual machines (VM). VMs are useful if you want to emulate an entire system, including a GUI Environment. However, they are not as lightweight, portable, or as fast as a docker container. This is because VM's can require a lot of CPU, RAM, and Memory to run. This makes a lot of VM's slow when the system they are running on does not have powerful enough hardware. But despite the demands from a VM, it is usually a faithful emulation of the OS so it makes for a good testing environment.

CTest

`CTest` is a testing tool used by CMake. It works by running test files or scripts that the user has created. Tests pass by default if they return 0 and fail if they return any other value/number.

To enable this tool in your CMakeLists.txt file add the following lines adjusted for your project:

```
enable_testing()
file(GLOB TEST_SOURCES "test/*.cpp")
foreach(TESTFILE ${TEST_SOURCES})
    message("-- Test for ${TESTFILE} created successfully")
    get_filename_component(TEST_NAME ${TESTFILE} NAME_WLE)

    add_executable(${TEST_NAME} ${TESTFILE})
    target_include_directories(${TEST_NAME} PUBLIC ${LIBXML2_INCLUDE_DIR})
    target_include_directories(${TEST_NAME} PUBLIC src)
    target_link_libraries(${TEST_NAME} PUBLIC ${LIBXML2_LIBRARY})
    add_test(NAME ${TEST_NAME} COMMAND ${TEST_NAME})
endforeach()
```

This will create a test program and test for each file in the specified directory. From there, after running `cmake` and either `ninja` or `make` you can run the following.

```
$ #For ninja generator
$ ninja test
$ #For make generator
$ make test
```

Docker-Compose

Docker-Compose is a tool that allows you to automate testing on multiple containers concurrently. In the Examples folder there is a file called docker-compose.yml that can be used to see how it works. By using the command `docker-compose up` on a properly configured file docker will run the commands set up in the file across many different builds.

General syntax for a `docker-compose.yml` is as follows.

```
#Version of docker-compose to use
version: "3.8"
```

```
# services defines what images and commands to run
services:
  #Name each container service
  ubuntu_2004:
    #Define the image to pull from
    image: ubuntu:20.04
    #Define all commands to run
    command: bash -c
      "apt -qq update;
      apt -qq upgrade -y >dev/null;
      apt -qq install figlet -y > dev/null;
      figlet DevOps"
```

To run docker-compose type the following into your terminal.

```
$ #config is used to make sure there are no errors in your docker-compose file
$ docker-compose config
$ #up is used to run all services
$ docker-compose up
$ #specify a service name to only run that service and disregard running other
services
$ docker-compose ubuntu_2004
```

To learn more about other aspects of docker-compose like `x-setup`, arguments, and environment variables, check out the documentation.

Shell Aliases

Aliases are incredibly useful when debugging, because they allow you to type out long complicated commands as one simple command. An alias works just like a function in many programming languages except that it doesn't accept arguments for the commands typed in.You can however call an alias and have a parameter after to have the commands act on that parameter. To create a temporary alias (one that only lasts as long as the terminal session is running) type the following.

```
$ #General Syntax
$ alias aliasName='shell commands'
$ #Example
$ alias rem='rm -rf'
$ #Running the Alias
$ rem Folder_to_be_Deleted
$ #Result: Folder and all files in the folder were deleted
```

When you type in the name for your alias during the terminal session, then the command(s) you assigned to the alias will be run.

Shell Functions

Another ability of the shell are shell functions. These are functions written in the shell's language (bash, zsh, fish) that can be used to automate almost any command. See below for an example.

```
$ #General Syntax
$ function funcName {
  #Bash code or commands here each new line has a ; at the end of it
}
$ #Example
$ function runD {
  docker run --workdir /build --mount
type=bind,source="$(pwd)",target=/$1,readonly -it $2;
}
$ #Running function
$ runD project ubuntu:latest
$ # Runs the docker commands with the directory mount name being the first
parameter and the docker image to use as the second parameter.
```

Installation

CPack

`CPack` is a tool that also comes with CMake that should be used to help package and install your code onto other systems. Of all the tools that comes with `CMake`, `cpack` is probably one of the easiest and most useful ones. To enable cpack, all you need to do is copy the following lines into your CMakeLists.txt.

```
include(CPack)
```

That is all you need. Now generally you want more than just the bare minimum, so here are some other useful commands to write in your CMakeLists.txt.

Stripping executables

```
# Strip Executable and shared libraries
set(CPACK_STRIP_FILES ON)
```

Adding copyright files

```
# Copyright file included in package
set(CPACK_RESOURCE_FILE_LICENSE ${CMAKE_SOURCE_DIR}/COPYING.txt)
install(FILES ${CMAKE_SOURCE_DIR}/COPYING.txt DESTINATION share/doc/srccomplexity
RENAME copyright)
```

Adding Descriptions

```
# Description included in package
set(CPACK_PACKAGE_DESCRIPTION_SUMMARY "Convert markdown to json, xml, and yaml
formats")
file(WRITE ${CMAKE_BINARY_DIR}/description.txt
"markturn Hi

")
```

Setting Dependencies

```
# Set Dependencies
set(CPACK_DEBIAN_PACKAGE_DEBUG OFF)
set(CPACK_DEBIAN_PACKAGE_SHLIBDEPS ON)
set(CPACK_DEBIAN_PACKAGE_GENERATE_SHLIBS ON)
```

Setting up ldconfig triggers

This one is generally used for when your build relies on a shared library

```
# Handle ldconfig
set(TRIGGERS_FILE ${CMAKE_CURRENT_BINARY_DIR}/triggers)
file(WRITE ${TRIGGERS_FILE} "activate-noawait ldconfig\n")
set(CPACK_DEBIAN_PACKAGE_CONTROL_EXTRA ${TRIGGERS_FILE})
```

Additional Settings

```
include(GNUInstallDirs)
set(CPACK_PACKAGE_DESCRIPTION_FILE ${CMAKE_BINARY_DIR}/description.txt)
set(CPACK_PACKAGE_NAME "markturn")
set(CPACK_PACKAGE_VERSION_MAJOR "1")
set(CPACK_PACKAGE_VERSION_MINOR "0")
set(CPACK_PACKAGE_VERSION_PATCH "0")
set(CPACK_PACKAGE_CONTACT "Alex K Karwowski <akk42@zips.uakron.edu>")
include(CPack)
```

Something else to note with `cpack` is that you can specify the type of package created. By running the following command, you can create packages for many different systems.

```
$ #Using -G specifies the type of package to create
$ cpack -G DEB
$ #DEB is for the Debian based OSs
```

```
$ cpack -G RPM
$ #RPM is for Fedora based OSs
```

Apt Install

`apt` and its commands like `apt install` is a powerful command line package manager tool for Ubuntu and other Debian based systems. `apt install` allows you to install packages (specifically debian packages) via the command line from an online "apt" repository. You can also use `apt install` to install Debian packages you've created via cpack or another packaging tool. To use `apt install` perform the following commands.

```
$ apt install -y g++
$ #This installs g++ from the apt repository
$ #The -y flag is optional and just confirms that Yes you want to install the
package
```

Systemd

`systemd` is a program installed in various Unix systems that allow you to schedule different "services" to run whenever the system starts up. These services can range from programs that constantly run in the background, to managing keyboard layouts, and more. To create a service for a program create a file in `/etc/systemd/system/` with the name of you service with a `.service` extension appended to it. For this example my service's name is `reaction.service`

```
[Unit]
Description=Reaction Bot Service
After=network.target
StartLimitIntervalSec=0

[Service]
Type=simple
Restart=always
RestartSec=1
User=root
ExecStart=/reaction-bot/reaction.py

[Install]
WantedBy=multi-user.target
```

The important lines are the `Restart=always`, the `ExecStart=/reaction-bot/reaction.py` and `User=root`. The first one, `Restart=always`, allows a program to always run on start and if for some reason it stops running, then it will restart. The second one, `ExecStart=/reaction-bot/reaction.py` tells `systemd` where to find the program to run. Finally the last one, `User=root` tells `systemd` what user privilege to run the program under and if it runs for a certain user on multi-user systems.