Preface

You will to use a Linux lab machine for this lab. For remote access, Use the 'ssh' command to get to tuxworld.usask.ca. Work on the first 3 parts of this lab in parallel. Let the computer do the work for you, so you don't have to sit and wait for the virtual machine to install.

Activities/Description

1. Multi-architecture Make

Make is an incredibly powerful tool for building *targets* that have a dependency tree. It will only execute targets that have a dependency that has changed. This section of the lab will guide you through building a basic Makefile that compiles executables for an Intel 64-bit Linux system, a virtual machine with the ARM architecture (cmpt332-arm), and a virtual machine with a PowerPC architecture (cmpt332-ppc). Whenever you feel like you've made a resonable step forward in progress, commit to git (at least every 15 minutes). This is a review of Lab 0, with some added features.

There are 3 source files: `sample-linux.c`, `linux-lib.c`, and `lab1.h`. **Do not edit these files**.

First start out by defining a couple of variables.

```
CC = gcc
CFLAGS = -g
CPPFLAGS = -std=gnu90 -Wall -pedantic
```

You should use the `CPPFLAGS` variable whenever the C pre-processor runs, and `CFLAGS` whenever the C compiler runs. `CC` is used to specify which compiler you want to use. When searching for include files and/or libraries, you must explicitly specify where the files you need are located. The -I and -L directives to the compiler/linker, respectively will be necessary.

You will also need variables/targets for the executables on the various architectures that are required.

The makefile must do the following:

- compile `sample-linux.c` and `linux-lib.c` for Linux into the appropriate .o files. **NEVER** compile an executable directly from source code. Always generate the intermediate .o file.

- archive `linux-lib.o` into a library .a file (use the `ar` command)
- link the `sample_linux.o`, pthreads library and the linux-lib library to a `sample-linux` executable (using the `-L` and `-l` directives).
- have a target for each architecture: x86_64, arm, and ppc.

When you're on a Linux/Intelx86 system, you typically only want to compile and link the Linux/Intelx86 files. NORMALLY, you can detect this with `uname`. Correspondingly, when on a Linux/ARM, Windows, or Linux/PPC architecture, you will need to link with the correct libraries. Linking against UBC PThreads will also need the linker to look inside the operating system-specific and machine architecture-specific subdirectory of `/student/cmpt332/pthreads/lib`. The names of these subdirectories were generated by `uname -s` and `uname -m` combined. The name of the library is `libpthreads.a`. To know which system you are running, there is likely an environment variable called $OS that is defined when you log in. Do NOT use this variable from your environment, because that is not defined for the instructor's environment. If you use $OS, explicitly define it inside the makefile.

Directory Organization

The files your Makefile produces could be organized into separate folders based on their architecture. As easy way to do this is have everything starting in a `build` directory then have `build/obj/ARCH`, `build/lib/ARCH`, `build/bin/ARCH`. This way when you clean with your Makefile you can just remove the build directory.

Make has documentation online and the manual can be found here or the reference document in the course notes under "Software Resources and Documentation".

For this course, we want to keep makefiles simple and straightforward. There will be no automatically generated makefiles (i.e. **cmake** is specifically not permitted) and the use of sophisticated makefile magic is discouraged unless you really know what you are doing. Check with a TA if you intend to get fancy with your makefiles. In this level of software development, we want to have explicit control of exactly what gets compiled and linked. If you have any questions about this, please ask.

Cross-Compiling

Cross compiling allows you to create executable files for an architecture that isn't the one you're currently using. We are focussing on compiling for two architectures:

1. Raspberry Pi (cmpt332-arm): ARM architecture machine (actually a Virtual Machine made to look like a Raspberry Pi). Nearly all Cell phones use the ARM architecture, and many laptops, tablets and other embedded systems are based on the ARM architecture.
2. PowerPC (cmpt332-ppc): "One of the original architecures for Apple hardware. It has since become a niche in personal computers, but remains popular for embedded and high-performance processors. Its use in the 7th generation of video game consoles and embedded applications provide an array of uses, including satellites, and the Curiosity and Perseverance rovers on Mars. In addition, PowerPC CPUs are still used in AmigaOne and third party AmigaOS 4 personal computers." (Wikipedia).

In this case, the suggestion to use `uname` will not work.

To do this, use separate command/rules to do the following (for each architecture):

- Create a new rule specifically for cross compiling for the RPi
- Use a new variable `CROSS_COMPILE=arm-linux-gnueabihf-`. This variable gives the prefix for the build tools for the other architecture. Here we can see that it is Linux for the ARM architecture. Specifically, you can then access gcc, cpp, etc.
- Compile `sample-raspi.o` for ARM architecture (or use the structure above). from `sample-linux.c`
- Compile `raspi-lib.o` from `linux-lib.c`
- Convert `raspi-lib.o` into a library `.a` file

- Generate (via linking) a `sample-linux-arm` executable from `sample-raspi.o` and your local library and the UBC Pthreads library. Use whatever make targets you wish so that the appropriate binaries are built from your make command. The marking script will only run `make`.
- Ensure that the binary is in the main level of your directory.
- Repeat the same steps for the PowerPC architecture, with the appropriate prefix for the compiler and the text `ppc` for the object and executables.

Now verify that the program runs on the ARM architecture by doing an 'ssh cmpt332-arm' and 'ssh cmpt332-ppc', respectively (Note: these machines are not accessible from outside campus; you need to do this in tuxworld or from a lab Linux box) and then executing the program there. Do not compile on the virtual machine. You **could** compile there, but the point is to be able to cross-compile. This is useful when the target machine is resource-challenged as these VMs are.n

2. Bash Scripting

Write a bash script called `run-lab1.bash` that that verifies the correct number of command-line arguments, and runs the appropriate binary, passing any command line arguments through to the executable. At first glance, this may seem unnecessary, but it will be necessary/useful in future labs/assignments where you are going to be compiling and running programs on different hardware architectures.

3. Virtual Machine

In this lab you will be installing an Ubuntu Server virtual machine, building and installing a bleeding edge version of python.

- First ssh into trux, and launch virtualbox

    - `$ ssh -Y trux`

    - `$ virtualbox &`

- Under 'Preferences' change the 'Default Machine Folder' from `/student/$NSID` to `/u2/cmpt332/$NSID`

- Create a new virtual machine named whatever you think is appropriate. Make sure to select Linux as the type, and Ubuntu (64 bit) as the version.

- Give the virtual machine 4GB of memory, and create a new virtual hard disk.

    - Select VDI for the Hard disk file type, and ensure that it is a fixed size of 50 GB and then hit the create button.

- Click on settings, System, and give yourself 4 Processor(s) to finish your vm initialization.

- Start the machine in headless mode.

    - You'll notice that no window appears, this is normal!

- Select the vm you created and click the green 'Show' arrow.

Virtualbox should now be asking for a start-up disk. Select the Ubuntu Server image from /u2/cmpt332/images. The display will likely look very strange at this point. From the View dropdown, once the machine is trying to install from the image, select 'Scaled Display' After hitting ok, you may need to reset the virtual machine.

Follow the on-screen instructions. You don't need to install openssh server or any 'server snaps'. Make sure you wait for updates to finish installing, then select 'Reboot', and hit `Enter` if prompted. You have now installed your virtual machine!

Note, if you don't allow updates to install before you reboot, you are running an untested configuration and the TAs will not be able to assist.

IMPORTANT: If you ever need to take a break from the lab, click on 'Machine' and select 'Detach GUI'. This will allow your vm to keep working while you are away. Do not close the virtual machine by clicking the window's close button, as this will shut down your vm.

Log in to your virtual machine and confirm that it is running by capturing the text from the execution of `uname -a`. Place this in the file `vm-verification.txt`.

4. Read the Field Manual! (Building Bleeding Edge Python from source)

Prior to starting, run `$ script python.txt`. This will create a text file of your terminal output for you to hand in.

First you want to clone the latest version of python's git repository.

```
$ git clone https://github.com/python/cpython.git
```

Now follow the build instructions from the Readme located here.

There are two ways you can do this: The painful way of repeatedly running ./configure and installing dependencies as you discover they are missing, or reading the entire build instructions section of the readme before starting. TAs will not assist you if you are using the painful method.

Once compilation is complete, make sure to run the tests, install your new version of Python, and finish off with a `$ python3 version` to show you have completed the exercise. Remember to stop the script with `ctrl+d` once you have finished this part of the lab.

Deliverables

A single tar file containing your `Makefile`, `run-lab1.bash`, a text file containing the output of your `git log` called `git-log.txt`, a text file containing the login and verification of your virtual machine installation `vm-verification.txt`, and the output of your python compilation `python.txt`.

Submission

Create a .tar file (using `tar -cvf`, double-check with `tar -tvf`) with the 5 files (no directories) in the next section and submit it to Lab 1.

General Guidelines:

- Unless otherwise specified, your name, NSID and Student number **must be at the beginning** of **each individual file** that you hand in. That is each source file, each documentation file, each Makefile, each testing output, and each log file.
- Submit files only. Never submit anything that is explicitly inside a directory, unless otherwise indicated.
- Never submit binaries or executables.
- Make sure that the lines in all text/code files are 80 characters wide at most. This is to facilitate the marker's convenience. If you have to indent such that this is necessary, refactor your code. Something is wrong with your design.
- Double check your .tar file before submission. Tar files may get accidentally garbled. Untar your file and verify its contents before submission.

- Do not use any compression method on your tar file. The file should be uncompressed (ending in ".tar" only). Canvas will enforce this. Under no circumstances should you make a file of another type (e.g. zip) and rename it as a tar file. This cannot be accepted and will not be tolerated.

Grading

- `Makefile`: 1 mark
- `run.bash`: 1 mark
- `git-log.txt`: 1 mark
- `vm-verification.txt`: 1 mark
- `python.txt`: 1 mark