# Outlab 5 : Build Tools & PyNetworking

Please refer to the general instructions and submission guidelines at the end of this document before submitting.

## P1. C you again

C was one the first human readable programming languages designed for people in all domains (business and scientific community). C is an imperative procedural language created by Dennis Ritchie. Unix and linux kernels are built in C (and assembly) language. Due to this C code runs faster than any other language. But still C was a very low level language, meaning that bigger **abstractions** were not readily available in it.

To deal with this Bjarne Stroustrup created C++ with objectives of speed and abstraction in mind. The core of his creation was something called a **class.** Thus C++ became one of the first object oriented languages.

C has a simple architecture and few paradigms which makes it simple to use. Remember the zen of python (*There should be one—and preferably only one—obvious way to do it.*) But due to the sheer flexibility of C++ (with several paradigms it posses) it became very difficult to maintain standards in bigger C++ projects.

One of the biggest misconceptions in this area is that C and C++ are the same language. C is not object oriented while C++ is. Acknowledge the difference.

Today we will learn about compiling & linking C/C++ code in various ways using raw commands, Makefile and CMake. This is one of the ways to gain insights into how exactly things work in C/C++ world.

Create a Makefile named **rawmake** (can be run using **make -f rawmake**) and fill it up as instructed in the below tasks. Source and Header files are provided in P1/ directory. Assume that **rawmake** file will be placed directly under P1/ directory. The dir structure after creating **rawmake** file will be as follows

```
P1/
├── helloworld.cpp
```

```
├── myengine
│   ├── myengine.cpp
│   └── myengine.hpp
├── mygame
│   └── mygame.cpp
├── rawmake
└── usespthread.cpp
```

## Task 1

Add a rule to compile **helloworld.cpp** to form **helloworld** executable.
Command run while checking: **make -f rawmake helloworld**

## Task 2

Add a rule to compile **usespthread.cpp** to form **usespthread** executable. This file uses pthread library used to create and manage threads. You will be using this a lot in your Operating Systems course.
In order to properly compile this file you need to special use a flag for g++.
Command run while checking: **make -f rawmake usespthread**

## Task 3

In this task we build a library using files **myengine.hpp** and **myengine.cpp.** There are actually two types of libraries categorized based on the way the library is linked to main file. Dynamic and Static libraries. This and This are good reads on compiling static and dynamic libraries. This is a good read which distinguishes one from the other.

Use the given resources to add rules to make dynamic and static libraries. Dynamic library created should have name **libMyEngineDynamic.so** and similarly static **libMyEngineStatic.a**
Command run while checking: **make -f rawmake libMyEngineDynamic.so**
Command run while checking: **make -f rawmake libMyEngineStatic.a**

## Task 4

In this task we try to install the above built libraries into the system (requires sudo permission). Add a PHONY rule named **installdynamic** which installs dynamic version (.so file) to **/usr/local/lib/** and corresponding headers to **/usr/local/include/**

Do the same for static version (.a file) by creating a rule **installstatic**
Command run while checking: **make -f rawmake installstatic**
Command run while checking: **make -f rawmake installdynamic**

These rules should have previously created libraries as dependencies, so that they are built first (if not already built) and then installed.

## Task 5

Now we use the installed libraries in **mygame.cpp**
Add a rule to compile **mygame.cpp** with the static library installed in the previous task and produce binary name **mygamestatic**.
Command run while checking: **make -f rawmake mygamestatic**

## Task 6

Add a rule to compile **mygame.cpp** with the dynamic library installed in the previous task and produce binary name **mygamedynamic**.
Command run while checking: **make -f rawmake mygamedynamic**

## Task 7

Also add a PHONY rule to **clean** all the generated intermediates and binaries (.o .a. .so and other binaries)
Command run while checking: **make -f rawmake clean**

## Task 8

Now your job is to do the same tasks using CMake.
To do this you need to create a **CMakeLists.txt** file (directly under P1/ directory) which generated Makefile to build and install the targets in the same manner as above tasks 1 - 4

1. **helloworld** binary from **helloworld.cpp**
2. **usespthread** binary from **usespthread.cpp**
3. **libMyEngineDynamic.so** and **libMyEngineStatic.a** libraries from **myengine.hpp** and **myengine.cpp**
4. Instead of installing two types of libraries separately, it should now install both of the libraries and the corresponding header file **myengine.hpp** to **/usr/local/lib/** and corresponding headers to **/usr/local/include/** respectively. The command used will be **sudo make install**
5. Observe that the makefile that **CMakeLists.txt** creates already contains PHONY rule clean which removes all the targets and intermediates generated

Command run while checking: (from P1 dir)

1. **mkdir build**
2. **cd build**
3. **cmake ..**
4. **make** (should create helloworld, usespthread, libMyEngineStatic.a, libMyEngineDynamic.so)
5. **sudo make install** (should install libMyEngineStatic.a, libMyEngineDynamic.so and header into specified paths appropriately)

## Task 9

Now we want to do tasks 5 and 6 using CMake. To do this create a **CMakeLists.txt** file in **P1/mygame** directory. This file should compile **mygame.cpp** with the static library installed in the previous task and produce binary name **mygamestatic** and similarly compile **mygame.cpp** with the dynamic library installed in the previous task and produce binary name **mygamedynamic**

Command run while checking: (from mygame dir)

1. **mkdir build**
2. **cd build**
3. **cmake ..**
4. **make** (should create **mygamestatic** and **mygamedynamic**)

The dir structure after creating **rawmake** file and two CMakeLists.txt files will be as follows

```
P1/
├── CMakeLists.txt                    (For task 8)
├── helloworld.cpp
├── myengine
│   ├── myengine.cpp
│   └── myengine.hpp
├── mygame4
│   ├── CMakeLists.txt                (For task 9)
│   └── mygame.cpp
├── rawmake                           (For tasks 1-7)
└── usespthread.cpp
```

Make sure the directory structure like shown above and submit the entire P1/ directory during submission.

# P2. Counter

## Task 1

Create **count.py** which does the following:

1. Fetches the content at the URL given below.
   https://www.cse.iitb.ac.in/page222
   The page shows the following content -
   - B.Tech - I
   - B.Tech - II
   - B.Tech - III
   - B.Tech - IV
   - …

   2. Opens each of the links and counts the number of students in each of the above categories.
   3. Stores the data to **count.csv** in the following format -

| *Category Name* | *No. of students* |
|---|---|
| *B.Tech - I* | *0* |
| *B.Tech - II* | *127* |
| *...* | *...* |

(Hint : Use requests, BeautifulSoup libraries)

## Task 2

Create **db.py** which does the following:

1. Creates a database **cse_students.sqlite** and populate it with the above data by reading the csv file, **count.csv**. Let the column names in the database be same as the field names in **count.csv**.
2. Create returnCount(...) function which takes as argument the category name and returns the corresponding count of students.
   The name of the category is to be read from STDIN.

(Hint: Use sqlite3 library)

## Task 3:

Create **getloc.py** which contains three functions:

1. Create function iss_location() which goes to the link mentioned below and returns current latitude and longitude of the International Space Station(ISS) : http://api.open-notify.org/iss-now.json
   (Hint: Use request/response along with json)

2. Create function pass_time() which takes longitude and latitude as arguments and returns duration(In minutes and seconds), and Date Time when the ISS passes over the given longitude/latitude. Use the below link to get this data:
   http://api.open-notify.org/iss-pass.json
   (Hint: Use request/response with get/post to send/receive values and datetime module to convert timestamp)
   Use 1 example of longitude/latitude and print the result. Read the longitude and latitude from STDIN.

3. Create function people_info() which goes to the below link and returns the number and names of the people currently in space: http://api.open-notify.org/astros.json

   Example:
   >$python getloc.py
   >Current Location of ISS:
   >Latitude : 1.4912
   >Longitude : -50.2963
   >Enter Details to know when ISS will pass over a location:
   >Latitude : 45.0
   >Longitude : -122.3
   >Date : 31/02/2019
   >Time : 14:20
   >For : 3 minutes and 45 seconds
   >People currently in space: 2
   >1. Neil Armstrong
   >2. Rakesh Sharma

```
P2/
├── count.py
├── count.csv
├── db.py
```

```
├── cse_students.sqlite
└── getloc.py
```

# P3. Understanding Make

## Task 1 - Tracking files

Build a code base structured as follows.

depend.h: declaration of a function
depend.cpp: implementation of the function
main.cpp: main function using the function

> Write a makefile named **Makefile** for the above code base that produces an executable **main** upon running *make* (exactly as shown in lecture on make on 14th Aug 2018).

Now add a comment to **depend.h** or *touch* it and run *make*. The build process is rerun even though it doesn't really need to in this case.

To deal with the above problem, make has an inbuilt feature which just updates the timestamps of all files that are to be rebuilt.
Write the command that does this in under a phony target **skiprebuild**.

This method is valid in general to avoid rebuilds when any arbitrary file is modified, but it heavily tampers the properties of the files.

In this particular situation we are only required to prevent rebuilds when the particular file, **depend.h** is changed. We could optimize the solution. We know that a target is built when the timestamps of its dependencies are earlier than that of the target.

> Write a series of bash commands (or one) under a phony target **skiprebuildh,** that inspects the timestamps of **depend.o**, **main.o** and updates the timestamp of **depend.h** to be *any time* prior to that of both the object files.

Verify that the script works as intended and only modifies the timestamp of **depend.h** and that it does in fact avoid future involcations of *make*.

Files to be submitted: depend.h, depend.cpp, main.cpp, Makefile

## Task 2 - Rebuilding

Download the source code for bash-4.4 from this location.
Extract, configure and compile the code according to instructions in **INSTALL** file. Use the default options. Install dependencies if necessary.

Bash is now available as an executable **bash**.

We wish to change the source code now and study how the rebuilding process works.

The file **execute_cmd.c** contains the error message displayed when a command is not found by bash (line number 5232). Change it to some other an appropriate message and save the file.
Now rebuild your code base.

> Inspect what targets are executed and make a list of the files rebuilt (referred to as 'tg') in the order they are built. Also mention the corresponding files (referred to as 'dp') that changed and triggered the execution of a target which produced that particular file (tg). Write this in a text file named **execution**.

**Format:**
*tg1: dp1 dp2 ...*
*tg2: dp1 dp2 ...*
...

You can try to parse the Makefile yourself and keep track of the changing files, but do not do this. Use features in *make*.

Your list should contain an "execute_cmd.o: execute_cmd.c" for obvious reasons. Curiously enough you won't find such a line in the Makefile. Read about pattern rules.

Additional Exercise: Implicit rules in make.

## Task 3 - Parallelism

Clean the bash directory using *make clean*.
Now time the execution of make.

***Usage:*** *time <command>*

There is some inherent parallelism in the way make functions.
If there are two targets that depend on the same file, they can potentially be run in parallel if one is not a dependency of the other.

Make implements this by creating multiple jobs using the option *-j*.

Clean your directory and run *make -j <nthreads>*. Try to not spawn too many threads. It might lead to worse performance (stick to ~ncores).

Verify that the process has in fact completed faster and that multiple cores were used.

Report both (real) times in a text file **execution_times** as shown in the example.

**Example:**
make: 28s
make -j: 32s

# General Instructions

- Make sure you know what you write, you might be asked to explain your code at a later point in time
- Your code may be tested on hidden test cases
- Grading may be done automatically, so please make sure you stick to naming conventions
- The deadline for this lab is **Monday, 20th August, 06:00.**

# Submission Instructions

After creating your directory, package it into a tarball
**`<rollno1>-<rollno2>-<rollno3>-outlab5.tar.gz`** in ascending order. Submit once only per team from the moodle account of smallest roll number.
The directory structure should be as follows (nothing more nothing less)

```
<rollno1>-<rollno2>-<rollno3>-outlab5
├── P1
│   ├── CMakeLists.txt                  (For task 8)
│   ├── helloworld.cpp
```

```
│   ├── myengine
│   │   ├── myengine.cpp
│   │   └── myengine.hpp
│   ├── mygame
│   │   ├── CMakeLists.txt              (For task 9)
│   │   └── mygame.cpp
│   ├── rawmake                              (For tasks 1-7)
│   └── usespthread.cpp
├── P2
│   ├── count.csv
│   ├── count.py
│   ├── cse_students.sqlite
│   ├── db.py
│   └── getloc.py
└── P3
    ├── depend.cpp
    ├── depend.h
    ├── execution
    ├── execution_times
    ├── main.cpp
    └── Makefile
```