ECE 252
Systems Programming and Concurrency
Laboratory Projects Manual

by

Yiqing Huang
Jeff Zarnett
Hassan Nahas
Murray Dunne
Maran Ma

Waterloo, Ontario, Canada

Contents

List of Tables vi

List of Figures vii

Preface 1

I Lab Administration 1

II Lab Projects 7

1 Introduction to Systems Programming in Linux Computing Environment 8

1.1 Introduction . . . 8

1.1.1 Objectives . . . 8

1.1.2 Topics . . . 8

1.2 Starter Files . . . 9

1.3 Pre-lab Preparation . . . 9

1.3.1 Basic Linux Commands Exercises . . . 9

1.3.2 Pre-lab Programming Exercise . . . 12

1.4 LAB Project Assignment . . . 13

1.4.1 Problem statement . . . 13

1.4.2 The findpng command . . . 13

1.4.3 The catpng command . . . 15

1.5 Deliverables . . . 19

1.5.1 Pre-lab deliverables . . . 19

1.5.2 Post-lab Deliverables . . . 19

1.6 Marking Rubric . . . 20

2 Multi-threaded Programming with Blocking I/O **21**

2.1 Objectives . . . 21

2.2 Starter Files . . . 21

2.3 Pre-lab Preparation . . . 22

2.4 Lab Assignment . . . 22

2.4.1 Problem Statement . . . 22

2.4.2 Requirements . . . 23

2.4.3 Man page of paster . . . 24

2.5 Programming Tips . . . 25

2.5.1 The libcurl API . . . 25

2.5.2 The pthreads API . . . 25

2.6 Deliverables . . . 26

2.6.1 Pre-lab deliverables . . . 26

2.6.2 Post-lab Deliverables . . . 26

2.7 Marking Rubric . . . 26

3 Interprocess Communication and Concurrency **27**

3.1 Objectives . . . 27

3.2 Starter Files . . . 27

3.3 Pre-lab Preparation . . . 28

3.4 Lab Assignment . . . 29

3.4.1 The Producer Consumer Problem . . . 29

3.4.2 Problem Statement . . . 30

3.5 Deliverables . . . 33

3.5.1 Pre-lab Deliverables . . . 33

3.5.2 Post-lab Deliverables . . . 33

3.6 Marking Rubric . . . 36

4 A Multi-threaded Web Crawler **37**

4.1 Objectives . . . 37

4.2 Starter Files . . . 37

4.3 Pre-lab Preparation . . . 38

4.4 Lab Assignment . . . 38
4.4.1 Problem Statement . . . 38
4.4.2 The findpng2 command . . . 39
4.4.3 Web crawling . . . 40
4.4.4 The HTTP . . . 41
4.4.5 Programming Tips . . . 42
4.5 Deliverables . . . 42
4.5.1 Pre-lab deliverables . . . 42
4.5.2 Post-lab Deliverables . . . 42
4.6 Marking Rubric . . . 44

5 Asynchronous I/O with cURL 45
5.1 Objectives . . . 45
5.2 Starter Files . . . 45
5.3 Pre-lab Preparation . . . 46
5.4 Lab Assignment . . . 46
5.4.1 Problem Statement . . . 46
5.4.2 The findpng3 command . . . 46
5.5 Deliverables . . . 48
5.5.1 Pre-lab Deliverables . . . 48
5.5.2 Post-lab Deliverables . . . 48
5.6 Marking Rubric . . . 49

III Software Development Environment Quick Reference Guide 50

1 Introduction to ECE Linux Programming Environment 51
1.1 ECE Linux Servers . . . 51
1.2 Connecting to Linux servers . . . 51
1.3 Basic Software Development Tools . . . 53
1.3.1 Editor . . . 54
1.3.2 C Compiler . . . 55
1.3.3 Debugger . . . 55
1.4 More on Development Tools . . . 56

1.4.1 How to Automate Build 56
1.4.2 Version Control Software 59
1.4.3 Integrated Development Environment 59
1.5 Man Page 59

A Forms **60**

References **62**

List of Tables

0.1 Project Deliverable, Weight of the Lab Distribution, Lab Tutorial and Deadlines in EST. 3

1.1 IHDR data field and value 18
1.2 Lab1 Marking Rubric 20

2.1 Lab2 Marking Rubric 26

3.1 Timing measurement data table for given (B, P, C, X, N) values. 35
3.2 Lab3 Marking Rubric 36

4.1 Timing measurement data table for given (T, M) values. 43
4.2 Lab4 Marking Rubric 44

5.1 Timing measurement data table for given (T, M) values. 49
5.2 Lab5 Marking Rubric 49

C1 Programming Steps and Tools 54

List of Figures

1.1 Image Concatenation Illustration 13

C1 MobaXterm Path on ECE Nexus Windows 10 Machines. 52

C2 MobaXterm Welcome Page. 52

C3 MobaXterm Welcome Page. 53

C4 Linux files on P drive, a network mapped drive. 54

Preface

Who Should Read This Lab Manual?

This lab manual is written for students who are taking Electrical and Computer Engineering (ECE) Systems Programming and Concurrency course ECE 252 at the University of Waterloo.

What is in This Lab Manual?

The first purpose of this document is to provide the descriptions of each laboratory project [1]. The second purpose of this document is a quick reference guide of the relevant development tools for completing laboratory projects. This manual is divided into three parts.

Part I describes the lab administration policies

Part II is a set of course laboratory projects as follows.

- Lab1: Introduction to systems programming in Linux computing environment
- Lab2: Multi-threaded concurrency programming with blocking I/O
- Lab3: Inter-process communication and concurrency control
- Lab4: Parallel web crawling
- Lab5: Single-threaded concurrency programming with asynchronous I/O

Part III is a quick reference guide of the Linux software development tools. We will be using the Ubuntu Linux operating system. Materials in this part should be self-studied before the lab starts. The main topics are as follows.

- Linux hardware environment
- Text editors
- Compiler

[1] We use lab project, lab, project interchangeably in this document

- Debugger
- Utility to automate build
- Utility for version control

Acknowledgments

We are grateful that Professor Patrick Lam shared his ECE 459 projects with us. Eric praetzel has provided continuous IT support, which makes the Linux computing environment available to our students.

We would like to sincerely thank our students who took ECE 252 and ECE 459 courses in the past few years. They provided constructive feedback every term to make the manual more useful to address problems that students would encounter when working on each lab assignment.

Part I

Lab Administration

Lab Administration Policy

Lab Format

The lab consists of five projects to be completed throughout the semester. Introductions for each lab are delivered as a recording on Learn, Q&A before starting the lab is available in the tutorial session of the lab weeks.

Group Lab Policy

- **Group Size.** All labs are done in groups of *two*. A group size of one is not recommended but it is permitted, e.g. if a group is split up. There is no workload reduction if you do the labs individually. Everyone in the group normally gets the same mark. Learn (https://learn.uwaterloo.ca) is used to sign up for groups and for grade compilation. After the sign-up deadline, students without a group will be assigned to random groups on Learn and will lose the lab sign-up mark. These groups cannot be changed except through the group split-up procedure outlined below. Grace day policy does not apply to the Group Sign-up deadline.
- **Group Split-up.** If you notice workload imbalance, try to solve it as soon as possible within your group, or split up the group as the last resort. Group split-up is only allowed once. You are allowed to join a one member group after the split-up. But you are not allowed to split up from the newly formed group again. We highly recommend everyone to stay with your group members as much as possible, for the ability to do team work will be an important skill in your career. Please choose your lab partners carefully. A copy of the code and documentation completed before the group split-up will be given to each individual in the group.

Deliverable	Weight	Lab Tutorial for Q&A	Tendative Deliverable Deadline
Group Sign-up	2	N/A	23:59 Sunday, May 12 2024
LAB 1	20	May 13	23:59 Friday, May 24 2024
LAB 2	18	May 27	23:59 Friday, June 7 2024
LAB 3	24	June 10	23:59 Friday, June 28 2024
LAB 4	18	July 1	23:59 Friday, July 12 2024
LAB 5	18	July 15	23:59 Friday, July 26 2024

Table 0.1: Project Deliverable, Weight of the Lab Distribution, Lab Session and Dead-lines in EST.

- **Group Split-up Deadline.** To split from your group for a particular lab project, you need to notify the lab instructor in writing and sign the group split up form (see Appendix). The Lab*n* (*n=1,2,3,4,5*) group split-up form needs to be submitted to the lab instructor by 23:59 EST Friday in the week that Lab*n* has a scheduled lab session. If you are late to submit the split-up form, then you need to finish Lab*n* as a group and split starting from Lab(*n+1*).

Lab Projects Grading and Deadline Policy

Labs are graded by lab TAs based on the rubric specified in each lab. The weight of each lab towards your final lab grade is listed in Table 0.1.

- **Lab Project Preparation and Due Dates.** Each lab has expected pre-lab activities that should be completed to the best ability of the group.
  The detailed deadlines of post-lab deliverables are displayed in Table 0.1. Please check Learn | Labs for the most up-to-date deadlines in case of deadline changes. Any deadline changes will also be communicated on Piazza.

- **Lab Assignment Submissions and Late Policy.** Submission of each lab will be through the lab's gitlab group https://git.uwaterloo.ca/ece252-s24/ (available after group sign-ups are processed). Each group will have its own consistent repository with a folder for each lab. Submission for a given lab will take the form of pushing to the group's repository before the deadline.
  - You may discuss ideas, design alternatives, and help other groups debug small fragments of code. However, each group must submit their own, independently developed code for each lab.

- Each group has 3 grace days[1] (no late penalty applied) for the term that can be used for any of L1-L5 deliverables. An on-line form will be provided to request grace days.
- If you run out of grace days, or choose not to use it, you can still submit a deliverable by the last day of lectures and recuperate up to `50%` of the deliverable's value.
- Absolute last day for any lab submission is the last day of lectures for the term, regardless of pending grace days.

- **Lab Re-grading.** After receiving your grading report (in your repo), you have 2 days to make a resubmit. There is no limitation on the number of lines of code change for the resubmit. Please note this is meant for correcting trivial oversights, not for rewriting the entire lab, hence the time constraint. An on-line form will be provided to request the regrade.

Lab Repeating Policy

For a student who repeats the course, labs need to be re-done with a new lab partner. Simply turning in the old lab code is not allowed. We understand that the student may choose a similar route to the solution chosen last time the course was taken. However it should not be identical. The labs will be done a second time, we expect that the student will improve the older solutions. Also the new lab partner should be contributing equally, which will also lead to differences in the solutions.

Note that the policy is course specific to the discretion of the course instructor and the lab instructor.

Lab Assignments Solution Internet Policy

It is not permitted to post your lab assignment solution source code or lab report on the internet freely for public to access. For example, it is not acceptable to host a public repository on GitHub that contains your lab assignment solutions. A warning with instructions to take the lab assignment solutions off the internet will be sent out upon the first offence. If no action is taken from the offender within twenty-four hours, then a lab grade zero will automatically be assigned to the offender.

Seeking Help on Lab Projects

- **Lab Help Sessions and Office hours** Scheduled lab sessions are the best opportunity for work time and seeking help. The Lab Instructor will be present in all sessions.

[1] Grace days are calendar days. Days in weekends and holidays are counted.

Additional lab sessions may be offered for each lab project and would be announced via Piazza. Office hours for weeks without lab sessions will be posted on Piazza.

- **Discussion Forum.** We recommend students to use the Piazza discussion forum to ask the teaching team questions instead of sending individual emails to lab teaching staff. For questions related to lab projects, our target response time is one business day prior to the deadline of the particular lab in question [2]. *There is no guarantee on the response time to questions of a lab that passes the submission deadline*.

- If a question requires teaching staff to look at a code fragment, please bring a laptop with necessary development software in-stalled.
  Please note that teaching staff will not debug student's program for the student. Debugging is part of the exercise of finishing a programming assignment. Teaching staff will be able to demonstrate how to use the debugger and provide case specific debugging tips. Teaching staff will not give direct solution to a lab assignment. Guidances and hints will be provided to help students to find the solution by themselves.

- **Additional Links.** The following links may be useful to you during the development of your lab assignments:
  1. MobaXterm download:
     https://download.mobatek.net/2022020030522248/MobaXterm Installer v20.2.zip
  2. Linux command-line tutorial:
     https://ubuntu.com/tutorials/command-line-for-beginners#1-overview
  3. User manual for the ddd debugger:
     https://www.gnu.org/software/ddd/manual/pdf/ddd.pdf
  4. Detailed description of the PNG file format:
     http://www.libpng.org/pub/png/spec/1.2/PNG-Structure.html
  5. More information about POSIX threads (pthreads):
     https://computing.llnl.gov/tutorials/pthreads/

[2] Our past experiences show that the number of questions spike when deadline is close.The teaching staff will not be able to guarantee one business day response time when workload is above average, though we always try our best to provide timely response.

6. Background on System V shared memory:
https://www.softprayog.in/programming/interprocess-communication-using-system-v

Part II

Lab Projects

Lab 1

Introduction to Systems Programming in Linux Computing Environment

1.1 Introduction

1.1.1 Objectives

This lab is to introduce system programming in a general Linux Development Environment at ECE Department. After finishing this lab, students will be able to

- apply basic Linux commands to interact with the Linux system through shell;
- apply standard Linux C programming tools for system programming and
- create a program to interact with Linux file systems by applying the relevant system and libray calls.

1.1.2 Topics

Concretely, the lab will cover the following topics:

- Basic Linux commands
- C programming toolchain including `gcc`, `make`, and `ddd`
- Linux manual pages
- Linux system calls and file I/O library calls to traverse a directory and perform read/write operations on selected files.

1.2 Starter Files

The starter files can be found publicly on https://git.uwaterloo.ca/ece252-s24/ece252_starter, and in your GitLab's group repo https://git.uwaterloo.ca/ece252-s24/. It contains the following sub-directories where we have example code and image files to help you get started:

- the cmd_arg demonstrates how to capture command line input arguments;
- the images contains some image files;
- the ls demonstrates how to list all files under a directory and obtain file types;
- the png_util provides a set of utility functions to process a PNG image file;
- the pointer demonstrates how to use pointers to access a C structure; and
- the segfault contains a broken program that has a segmentation fault bug, which you will debug in the last Pre-lab exercise.

Using the code in the starter files is permitted and will not be considered as plagiarism.

1.3 Pre-lab Preparation

Read the Introduction to ECE Linux Programming Environment supplementary material in Part III Chapter 1. Do the pre-lab exercises in Section 1.3.1. Do the pre-lab programming exercise (see 1.3.2).

1.3.1 Basic Linux Commands Exercises

These pre-lab exercises are to practice some basic commands on Linux.

1. Use the MobaXterm to login onto `eceubuntu.uwaterloo.ca`. You are now inside the Linux shell and in your home directory. The home directory usually has a path name in the format of `/home/username`, where username normally is your UWID. For example, a user with UWID of `jsmith` has a home directory of `/home/jsmith`.

2. Use the `pwd` command to print the full filename of the current working directory. You should see your home directory name printed on the screen. For example: `/home/jsmith`.

3. Use the `echo $HOME` command to print your home directory path name. You will notice that the output matches the `pwd` output of exercise 2.

4. Use the `env` command to list all the environment variables and their values. Note that `HOME` is one of the many environment variables.

5. One important environment variable is `PATH`. It specifies a set of directories the system searches for executible programs. Use `echo $PATH` to see your `PATH` environment variable setting.

6. Execute command `which ls` to locate the directory the ls command is in. You will notice the directory is listed in `PATH` environment variable. When you issue a command and get an error message of "command not found", it means the command cannot be found after searching all the directories listed in `PATH` environment variable. A commonly seen error is that a command in your current working directory gives you "command not found" error. This is normally due to the fact that the current working directory `.` or `./` is not in the PATH. Consequently you need to add the path to the command name for the system to know where the command is. For example `./a.out` tells the system to run the command `a.out` located in the current working directory.

7. Use the `ls` command to list all files in your current working directory.

8. Read the online manual of the `ls` command by issuing `man ls` command to the shell. Find out from the manual what options `-l`, `-a` and `-la` do. Execute the `ls` command with these three options and see the execution results.

9. Create a directory as the work space of labs under your home directory. Name the newly created directory as `labs`. Read the man page of the command `mkdir` to see how to do it.

10. Change directory to the newly create directory of `labs`. Read the man page of command `cd` to find out how to change directory.

11. Clone the ece252 lab repository by using the command:
    `git clone https://git.uwaterloo.ca/ece252-s24/ece252_starter`
    A new directory named `ece252` will be created. It has starter code of ECE252 labs.

12. [Please skip this step until group sign-ups are processed ~Sept 19th] Clone your group's repository using the SSH or HTTPS urls
    `git clone "REPO URL FROM GITLAB"`.
    A new directory named after your group will be created. It has starter code of ECE252 labs. You should use this for your lab projects' work and submissions.

13. Read the man page of the `find` command by issuing `man find` command to the shell. Read what the `-name` option does. Use `find` with the `-name` option to find all the files with .png file extension in the `$HOME/labs/ece252` directory.

14. Change directory to where the `WEEF_1.png` is. Use `file WEEF_1.png` command to obtain the file type and image properties such as dimensions and bit depth.

15. Use `file` command to obtain the file type information of `Disguise.png`. You should see that this is not an image file though the file exension is .png. Use a text editor to open the file and see the contents. This exercise is to show you that the `file` command does not obtain the file type information based on the file extension. It looks into the contents of file to extract the file type information [1].

16. You can use the command `display` to display an image. For example, to view the `WEEF_1.png` file, use the command `display WEEF_1.png`.

17. Execute `cat red-green-16x16.png` command and you will notice the output are gibberish funny characters. This is because cat displays plain text file in a human readable form, not a binary file. The PNG image file is a binary file. Both vi and emacs have hexadecimal mode which displays the bytes in binary files in a hexadecimal format. They are pretty good for small size binary files. There are also few linux commands that perform hex dump of a file. The `xxd` is one of them. Try `xxd red-green-16x16.png` and see the output. Other similar commands include `od` and `hexdump`. Refer to the man pages of these commands for detailed usage instructions.

18. The `pngcheck` command test PNG image files for corruption. Some image viewers are able to display the image even part of the data are corrupted. An image can be opened by an image viewer does not guarantee it is not corrupted. You will notice that the `display` command will not display a corrupted PNG image file on Linux. But the same corrupted image file most likely can be displayed by Windows Paint program. Execute the following two commands and see what pngcheck output tells you.
    - `pngcheck red-green-16x16.png`
    - `pngcheck red-green-16x16-corrupted.png`

19. Images are binary files. To compare two binary files, we can use the `cmp` command. Read the man page of the `cmp` command and especially pay attention to the `-l` option. Use the `cmp` command with the `-l` option to find out which bytes in `red-green-16x16.png` and `red-green-16x16-corrupted.png` are different. Another tool is `vbindiff`, which displays and compares hexadecimal file(s).

20. Use gdb or ddd to run the program under /lab1/starter/segfault directory of the starter code. When the code generates a segmentation fault inside the

[1] A file has a magic number to indicate its type. The magic number is a sequence of bytes usually appearing near the beginning of the file. The file command checks the magic number. The PNG file's magic number is 89 50 4E 47 in hexadecimal, which is .PNG in ASCII.

debugger, run the gdb command `where` to see the stack trace and fix the segmentation fault problem of the code.

1.3.2 Pre-lab Programming Exercise

We will write some small pieces of code that can be used in the final lab project assignment code. We will create a command line program named `pnginfo` that prints the dimensions of a valid PNG image file and an error message to the standard output if the input file is not a PNG file or is a corrupted PNG file. The command takes one input argument, which is the path name of a file. Both absolute path name and relative path name are accepted. For example, command `./pnginfo WEEF_1.png` will output the following line:

```
WEEF_1.png: 450 x 229
```

If the input file is not a PNG file. Then output an error message. For example, command `./pnginfo Disguise.png` will output the following line:

```
Disguise.png: Not a PNG file
```

If the input file is a PNG file, but certain chunks has CRC error. That is the CRC value in the chunk does not match the CRC value computed by your program, then the program output something similar as what pngcheck does. For example, command `./pnginfo red-green-16x16-corrupted.png` will output the following line:

```
red-green-16x16-corrupted.png: 16 x 16
IDAT chunk CRC error: computed 34324f1e, expected dc5f7b84
```

You will find starter files under png_util directory are helpful. To make our pre-lab code reusable in the final lab, we will create two functions. One is `is_png()` which takes eight bytes and check whether they match the PNG image file signature. Another function is `get_data_IHDR()` which extracts the image meta information including height and width from a PNG file IHDR chunk. You are free to design the signatures of these two functions so that they will be re-used in your lab1 final solution and future labs 2 and 3 solutions[2]. However in `lab_png.h` you will find some existing function prototypes that we put there to show one possible function prototype design. Feel free to modify these function prototypes to fit your own design. For computing the CRC of a sequence of bytes, the starter file already provides the crc.c and crc.h and the main.c that demos how to call the crc function to do the computation.

[2] lab2 and lab3 are based on the code of lab1

1.4 LAB Project Assignment

1.4.1 Problem statement

You are given a directory under which some files are PNG images and some files are not. The directory may contain nested sub-directories[3]. All valid PNG images under the given directory are horizontal strips of a bigger whole image. They all have the same width. The height of each image might be different. The PNG images have the naming convention of `*_N.png`, where `N` is the image strip sequence number and `N=0, 1, 2, ...`. However a file with .png or .PNG extension may not be a real PNG image file. You need to located all the real PNG image files under the given directory first. Then you will concatenate these horizontal strip images sequentially based on the sequence number in the file name to restore the original whole image. The sequence number indicates the order the image should be concatenated from top to bottom. For example, file `img_1.png` is the first horizontal strip and `img_2.png` is the second horizontal strip. To concatenate these two strips, the pixel data in `img_1.png` should be followed immediately by the pixel data in `img_2.png` file. Figure 1.1 illustrates the concatenation order.

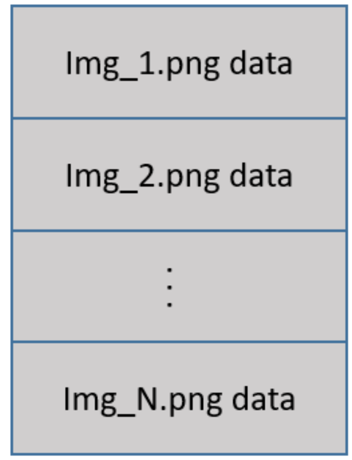


Figure 1.1: Image Concatenation Illustration

To solve the problem, first you will create a tool named `findpng` to search the given directory hierarchy to find all the real PNG files under it. Secondly you will create an image data concatenation tool named `catpng` to concatenate pixel data of a set of PNG files to form a single PNG image file. The `catpng` only processes PNG images with the same width in dimension.

1.4.2 The findpng command

The expected behaviour of the `findpng` is given in the following manual page of the command.

[3] A nested sub-directory is a sub-directory that may contain many layers of sub-directories.

Man page of findpng

NAME

findpng - search for PNG files in a directory hierarchy

SYNOPSIS

findpng DIRECTORY

DESCRIPTION

Search for PNG files under the directory tree rooted at DIRECTORY and return the search results to the standard output. The command does not follow symbolic links.

OUTPUT FORMAT

The output of search results is a list of PNG file relative path names[4], one file pathname per line. The order of listing the search results is not specified. If the search result is empty, then output "findpng: No PNG file found".

EXAMPLES

findpng .

Find PNG of the current working directory. A non-empty search results might look like the following:

```
lab1/sandbox/new_bak.png
lab1/sandbox/t1.png
png_img/rgba_scanline.png
png_img/v1.png
```

It might also look like the following:

```
./lab1/sandbox/new_bak.png
./lab1/sandbox/t1.png
./png_img/rgba_scanline.png
./png_img/v1.png
```

An empty search result will look like the following:

```
findpng: No PNG file found
```

[4]It is relative to the command input directory path name

Searching PNG files under a given directory

UNIX file system is organized as a tree. A file has a type. Three file types that this assignment will deal with are regular, directory and symbolic link. A PNG file is a regular file. A directory is a directory file. A link created by `ls -n` is a symblic link. Read the section 2 of `stat` family system calls man page for information about other file types. The `ls/ls_ftype.c` in the starter code gives a sample program to determine the file type of a given file. Note that the `struct dirent` returned by the `readdir()` has a field `d_type` that also gives the file type information. However it is not supported by all file system types. We will be using eceubuntu machines to test your submission. If you want to use the `d_type` in your code, make sure you test its behaviour on eceubuntu machines.

To search all the files under a given directory and its subdirectories, one need to traverse the given directory tree to its leaf nodes. The library call of `opendir` returns a directory stream for `readdir` to read each entry in a directory. One need to call `closedir` to close the directory stream once operations on it is completed. The control flow is to go through each entry in a directory and check the file type. If it is a regular file, then further check whether it is a PNG file by comparing the first 8 bytes with the PNG file header bytes (see Section 1.4.3). If it is a directory file, then you need to check files under the sub-directory and repeat what you did in the parent directory. The `ls/ls_fname.c` in the starter code gives a sample program that lists all file entries of a given directory.

Always check the man page of the systems calls and library calls for detailed information.

1.4.3 The catpng command

The expected behaviour of the `catpng` is given in the following manual page of the command.

Man page of the catpng

NAME

catpng - concatenate PNG images vertically to a new PNG named all.png

SYNOPSIS

catpng [PNG_FILE]...

DESCRIPTION

Concatenate PNG_FILE(s) vertically to all.png, a new PNG file.

OUTPUT FORMAT

The concatenated image is output to a new PNG file with the name of all.png.

EXAMPLES

```
catpng ./img1.png ./png/img2.png
```

Concatenate the listed PNG images vertically to all.png.

File I/O

There are two sets of functions for file I/O operations under Linux. At system call level, we have the *unbufferred I/O* functions: `open` , `read` , `write` , `lseek` and `close` . At library call level, we have standard I/O functions: `fopen` , `fread` , `fwrite` , `fseek` and `fclose` . The library is built on top of unbufferred I/O functions. It handles details such as buffer allocation and performing I/O in optimal sized chunks to minimize the number of `read` and `write` usage, hence is recommended to be used for this lab.

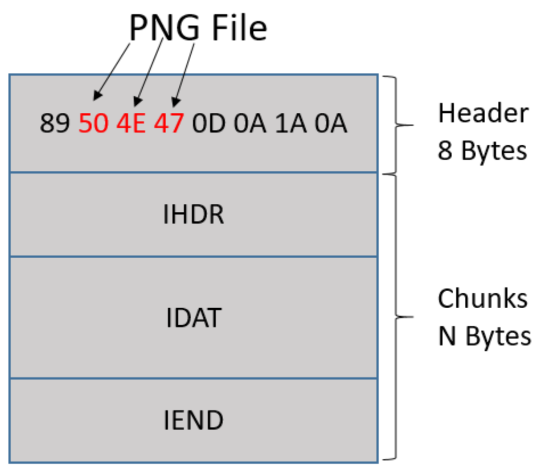
The `fopen` returns a FILE pointer given a file name and the mode. A PNG image file is a binary file, hence when you call `fopen` , use mode " `rb` " for reading and " `wb+` " for reading and writing, where the "b" indicates it is a binary file that we are opening. Read the man page of `fopen` for more mode options.

After the file is opened, use `fread` to read the number of bytes from the stream pointed by the FILE pointer returned by `fopen` . Each opened file has an internal state of file position indicator. The file position indicator sets to the beginning of the file when it is just opened. The `fread` operation will advance the file position indicator by the number of bytes that has been read from the file. The `fseek` sets the file position indicator to the user specified location. The `fwrite` writes user specified number of bytes to the stream pointed by the FILE pointer. The file position indicator also advances by the number of bytes that has been written. It is important to call `fclose` to close the file stream when I/O operations are finished. Failure to do so may result in incomplete files.

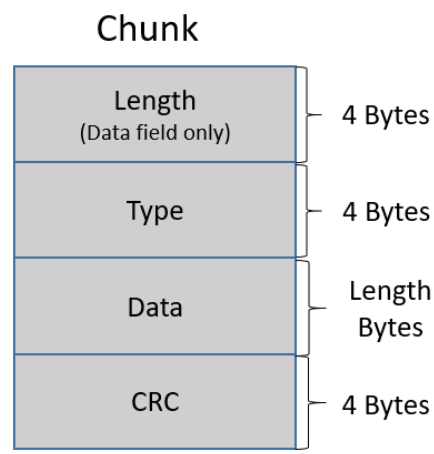
The man pages of the standard I/O library is the main reference for details including function prototypes and how to use them.

PNG File Format

In order to finish this assignment, one need to have some understanding of the png file format and how an image is represented in the file. One way to store an image is to use an array of coloured dots referred to as *pixels*. A row of pixels within an image is called a *scanline*. Pixels are ordered from left-to-right within each scanline. Scanlines appear top-to-bottom in the pixel array. In this assignment, each pixel is represented as four 8-bit [5] unsigned integers (ranging from 0 to 255) that specify the red, green, blue and alpha intensity values. This encoding is often referred to as the RGBA encoding. RGB values specify the colour and the alpha value specifies the opacity of the pixel. The size of each pixel is determined by the number of bits per pixel. The dimensions of an image is described in terms of horizontal and vertical pixels.



(a) PNG File Format



(b) PNG Chunk Format

PNG stands for “Portable Network Graphics”. It is a computer file format for storing, transmitting and displaying images[1]. A PNG file is a binary file. It starts with an 8-byte header followed by a series of chunks. You will notice the second, third and fourth bytes are the ASCII code of ’P’, ’N’ and ’G’ respectively (see Figure 1.2(a)).

The first chunk is the IHDR chunk, which contains meta information of the image such as the dimensions of the pixels. The last chunk is always the IEND chunk, which marks the end of the image datastream. In between there is at least one IDAT chunk which contains the compressed filtered pixel array of the image. There are other types chunks that may appear between IHDR chunk and IEND chunk. For all the PNG files we are dealing with in this assignment, we use the format that only has one IHDR chunk, one IDAT chunk and one IEND chunk (see Figure 1.2(a)).

Each chunk consists of four parts. A four byte length field, a four byte chunk type code field, the chunk data field whose length is specified in the chunk length field,

[5] Formally, we say the image has a bit depth of 8 bits per sample.

and a four byte CRC (Cyclic Redundancy Check) field (see Figure 1.2(b)).

The length field stores the length of the data field in bytes. PNG file uses *big endian* byte order, which is the network byte order. When we process any PNG data that is more than one byte such as the length field, we need to convert the network byte order to host order before doing arithmetic. The `ntohl` and `htonl` library calls convert a 32 bit unsigned integer from network order to host order and vice versa respectively.

The chunk type code consists four ASCII character. IHDR, IDAT and IEND are the three chunk type code that this assignment involves with.

The data field contains the data bytes appropriate to the chunk type. This field can be of zero length.

The CRC field calculates on the proceeding bytes in the type and data fields of the chunk. Note that the length field is not included in the CRC calculation. The `crc` function under the `png_util` starter code can be used to calculate the CRC value. The IHDR chunk data field has a fixed length of 13 bytes and they appear in the order as shown in Table 1.1. Width and height are four-byte unsigned integers giving the image dimensions in pixels. You will need these two values to complete this assignment. Bit depth gives the number of bits per sample. In this assignment, all images have a bit depth of 8. Colour type defines the PNG image type. All png images in this assignment have a colour type of 6, which is truecolor with alpha (i.e. RGBA image). The image pixel array data are filtered to prepare for the next step of compression. The Compression method and Filter method bytes encode the methods used. Both only have 0 values defined in the current standard. The Interlace method indicates the transmission order of the image data. 0 (no interlace) and 1 (Adam7 interlace) are the only two defined. In this assignment, all PNG images are non-interlaced. Table 1.1 Value column gives the typical IHDR values the PNG images you will be processing.

Name	Length	Value
Width	4 bytes	N/A
Height	4 bytes	N/A
Bit depth	1 byte	8
Colour type	1 byte	6
Compression method	1 byte	0
Filter method	1 byte	0
Interlace method	1 byte	0

Table 1.1: IHDR data field and value

The IDAT chunk data field contains compressed filtered pixel data. For each scanline, first an extra byte is added at the very beginning of the pixel array to indicate the filter method used. Filtering is for preparing the next step of compression. For example, if the raw pixel scanline is 4 bytes long, then the scanline after applying filter will be 5 bytes long. This added one byte per scanline will help to achieve better compression result. After all scanlines have be filtered, then the data are compressed according to the compression method encoded in IHDR chunk. The compressed data stream conforms to the zlib 1.0 format.

The IEND chunk marks the end of the PNG datastream. It has an empty data field.

Concatenate the pixel data

To concatenate two horizontal image strips, the natural way of thinking is to start with the pixel array of each image and then concatenate the two pixel arrays vertically. Then we apply the filter to each scanline. Lastly we compress the filtered pixel array to fill the data field of the new IDAT chunk of the concatenated image. However a simpler way exists. We can start with the filtered pixel data of each image and then concatenate the two chunks of filtered pixel data arrays vertically, then apply the compression method to generate the data field of the new IDAT chunk.

How do we get filtered pixel data from a PNG IDAT chunk? Recall that the data field in IDAT chunk is compressed data that conforms to zlib format 1.0. We can use zlib functions to uncompress(i.e. inflate) the data. The `mem_inf` in the starter code takes in memory compressed(i.e. deflated) data as input and returns the uncompressed data to a another memory location. For each IDAT chunk you want to concatenate, call this function and stack the returned data in the order you wish and then you have the concatenated filtered pixel array. To create an IDAT chunk, we need to compress the filtered pixel data. The `mem_def` function in the starter code uses the zlib to compress (i.e. deflate) the input in memory data and returns the deflated data. The `png_util` directory in the starter code demos how to use the aforementioned two functions.

To create a new PNG for the concatenated images, IHDR chunk also needs to have the new dimension information of the new PNG file. The rest of the fields of IHDR chunk can be kept the same as one of the PNG files to be concatenated. In this assignment, we assume that `catpng` can only process PNG files whose IHDR chunks only differ in the height field. So the new image will have a different height field, the rest of fields are the same as the input images.

1.5 Deliverables

1.5.1 Pre-lab deliverables

None.

1.5.2 Post-lab Deliverables

The lab is submitted by pushing to the group's gitlab repository before the deadline.

- Put the entire source code with a Makefile under the directory lab1. The Makefile default target includes `findpng`, and `catpng`. That is command `make` should generate the aforementioned two executable files. We also expect that command `make clean` will remove the object code and the default target. That is the `.o` files and the two executable files should be removed.

- Use `git add` to add the files to the git repo.

- Use `git commit` to commit the changes. The commit message should be ”Lab 1 Submission”. The final such commit will be considered.

- Use `git push` to push the changes to your repo, thereby performing the actual submission. Without pushing, a submission is not made.

1.6 Marking Rubric

Points	Sub-points	Description
100		Post-lab
	10	clean project files organization
	5	Makefile correctly builds and cleans
	35	Implementation of `findpng`
	50	Implementation of `catpng`

Table 1.2: Lab1 Marking Rubric

Table 1.2 shows the rubric for marking the lab. Note that if your code generates a segmentation fault, the maximum lab grade you can achieve is 60/100.

Lab 2

Multi-threaded Programming with Blocking I/O

2.1 Objectives

This lab is to learn about and gain practical experience in multi-threaded programming in a general Linux environment. A single-thread implementation using blocking I/O to request a resource across the network is provided. Students are asked to reduce the latency of this operation by sending out multiple requests simultaneously to different machines by using the `pthreads` library. After this lab, students will have a good understanding of

- how to design and implement a multi-threaded program by using the `pthreads` library; and
- the role mutli-threading plays in reducing the latency of a program.

2.2 Starter Files

The starter files for Lab 2 contains the following sub-directories where we have example code to help you get started:

- the cURL demonstrates how to use cURL to fetch an image segment from the lab web server;
- the fn_ptr demonstrates how C function pointers work;
- the getopt demonstrates how to parse command line options;
- the pthreads demonstrates how to create two threads where each thread takes multiple input parameters and return multiple values; and

- the times provides helper functions to profile program execution times.

Using the code in the starter files is permitted and will not be considered as plagiarism.

2.3 Pre-lab Preparation

Read the entire lab2 manual to understand what the lab assignment is about. Build the starter code and run the executables. Work through the code and understand what they do and how they work.

2.4 Lab Assignment

2.4.1 Problem Statement

In the previous lab, a set of horizontal strips of a whole PNG image file were stored on a disk and you were asked to restore the whole image from these strips. In this lab, the horizontal image segments are on the web. I have three 400×300 pictures (whole images) on three web servers. When you ask a web server to send you a picture, the web server crops the picture into fifty 400×6 equally sized horizontal strips [1]. The web server assigns a sequence number to each strip from top to bottom starting from 0 and increments by 1 [2]. Then the web server sleeps for a random time and then sends out a random strip in a simple PNG format that we assumed in lab1. That is the horizontal strip PNG image segment consists of one IHDR chunk, one IDAT chunk and one IEND chunk (see Figure 1.2(a)). The PNG segment is an 8-bit RGBA/color image (see Table 1.1 for details). The web server uses an HTTP response header that includes the sequence number to tell you which strip it sends to you. The HTTP response header has the format of "X-Ece252-Fragment: M" where $M \in [0, 49]$. To request a random horizontal strip of picture N, where $N \in [1, 3]$, use the following URL: `http://machine:2520/image?img=N`, where

`machine` is one of the following:

- `ece252-1.uwaterloo.ca`,
- `ece252-2.uwaterloo.ca`, and
- `ece252-3.uwaterloo.ca`.

For example, when you request data from the following URL:
`http://ece252-1.uwaterloo.ca:2520/image?img=1`,

[1] Each image segment will have a size less than 8KB.

[2] The first horizontal strip has a sequence number of 0, the second strip has a sequence number of 1. The sequence number increments by 1 from top to bottom and the last strip has a sequence number of 49.

you may receive a random horizontal strip of picture 1. Assume this random strip you receive is the third horizontal strip (from top to bottom of the original picture), the received HTTP header will contain "X-Ece252-Fragment: 2". The received data will be the image segment in PNG format. You may use the browser to view a random horizontal strip of the PNG image the server sends. You will notice the same URL displays a different image strip every time you hit enter to refresh the page. Each strip has the same dimensions of 400×6 pixels and is in PNG format.

Your objective is to request all horizontal strips of a picture from the server and then concatenate these strips to restore the original picture. Because every time the server sends a random strip, if you use a loop to keep requesting a random strip from a server, you may receive the same strip multiple times before you receive all the fifty distinct strips. Due to the randomness, it will take a variable amount of time to get all the strips you need to restore the original picture.

2.4.2 Requirements

Using the `pthreads` library, design and implement a threaded program to request all image segments from a web server by using blocking I/O and concatenate these segments together to form the whole image.

The provided starter code `main_write_header_cb.c` under `cURL` directory is a single-threaded implementation which uses `libcurl` blocking I/O function `curl_easy_peform()` to fetch one random horizontal strip of picture 1 from one of the web servers into memory and then output the received image segment to a PNG file. Your program should repeatedly fetch the image strips until you have them all. Recall because every time you get a random strip, the amount of time to get all the fifty distinct strips of a picture varies.

A very inefficient approach is to use a single-threaded loop to keep fetching until you get all fifty distinct strips of a picture and paste them together. You will notice the blocking I/O operation is the main cause of the latency. Your program will be blocked while each time waiting for the `curl_easy_perform()` to finish. One way to reduce the latency of this operation is to send out multiple blocking I/O requests simultaneously (to different machines) by using `pthreads`. You will use this approach to reduce the latency in this lab [3].

Your program should create as many threads as specified by the user command line input, and distribute the work among the three provided servers. Make sure all of your library (standard glibc and libcurl) calls are *thread-safe* (for glibc, e.g. `man 3 printf` to look at the documentation). Name your executable as `paster`. The behaviour of the command `paster` is given in the following section.

The provided three pictures on the server are for you to test your program. Your program should work for all these pictures. You may want to reuse part of your lab1 code to paste the received image segments together.

[3] Asynchronous I/O is another method to reduce the latency and we will explore it in lab5.

2.4.3 Man page of paster

NAME

paster - pasting downloaded png files together by using multiple threads and blocking I/O through libcurl.

SYNOPSIS

paster [OPTION]...

DESCRIPTION

With no options, the command retrieves all horizontal image segments of picture 1 from http://ece252-1.uwaterloo.ca:2520/image?img=1 and paste all distinct segments received from top to bottom in the order of the image segment sequence number. Output the pasted image to disk and name it all.png.

-t=NUM

create NUM threads simultaneously requesting random image segments from multiple lab web servers. When this option is not specified, assumes a single-threaded implementation.

-n=NUM

request a random image segment of picture NUM from the web server. Valid values are 1, 2 and 3. Default value is set to 1.

OUTPUT FORMAT

The concated image is output to a PNG file with the name of all.png.

EXAMPLES

```
paster -t 6 -n 2
```

Use 6 threads to simultaneously download all image segments of picture 2 from multiple web servers and concatenate these segments to restore picture 2. Output the concatenated picture to disk and name it all.png.

EXIT STATUS

paster exits with status 0 upon success and nonzero upon error.

2.5 Programming Tips

2.5.1 The libcurl API

Though the image segment download code using `libcurl` is provided, familiarizing yourself with the libcurl API will help you understand the provided code. The libcurl documentation URL is https://curl.haxx.se/libcurl. The man page of each function in the libcurl API can be found at URL https://curl.haxx.se/libcurl/c/allfuncs.html.

Note the provided example `cURL` code downloads the received image segment to memory and then output the data in memory to a PNG file. The output to a PNG file is just to make it easier for you to view the downloaded image segment to help you understand the example code. However your `paster` program does not need to output each segment received to a file. An efficient way (i.e. without unnecessary file I/O) is to directly use the received image segment data in memory instead of outputting the data to a file first and then reading the data back from file into memory.

Thread Safety

Libcurl is thread safe but there are a few exceptions. The man page of libcurl-thread(3) (see https://curl.haxx.se/libcurl/c/threadsafe.html) is the ultimate reference. We re-iterate key points from libcurl manual that are relevant to this lab as follows:

- The same libcurl handle should not be shared in multiple threads.
- The libcurl is thread safe but does not have internal thread synchronization mechanism. You will need to take care of the thread synchronization.

2.5.2 The pthreads API

The `pthreads(7)` man page gives an overview of POSIX threads and should be read. The SEE ALSO section near the bottom of the man page lists functions in the API. The man pages of `pthread_create(3)`,`pthread_join(3)` and `pthread_exit(3)` provide detailed information of how to create, join and terminate a thread.

The pthread Memory Leak Bug

There is a known memory leak bug related to `pthread_exit()`. Please refer to https://bugzilla.redhat.com/show_bug.cgi?id=483821 for details. Using `return()` instead of `pthread_exit()` will avoid the memory leak bug.

2.6 Deliverables

2.6.1 Pre-lab deliverables

None.

2.6.2 Post-lab Deliverables

Create a multi-threaded implementation of the `paster` command. The lab is submitted by pushing to the group's gitlab repository before the deadline.

- Put the entire source code with a Makefile under the directory lab2. The Makefile default target is `paster`. That is command `make` should generate the aforementioned executable file. We also expect that command `make clean` will remove the object code and the default target. That is the `.o` files and the executable file should be removed.
- Use `git add` to add the files to the git repo.
- Use `git commit` to commit the changes. The commit message should be "Lab 2 Submission". The final such commit will be considered.
- Use `git push` to push the changes to your repo, thereby performing the actual submission. Without pushing, a submission is not made.

2.7 Marking Rubric

Points	Sub-points	Description
100		Post-lab
	10	clean project files organization
	5	Makefile correctly builds and cleans
	85	Implementation of multi-threaded `paster`

Table 2.1: Lab2 Marking Rubric

Table 2.1 shows the rubric for marking the lab.

Lab 3

Interprocess Communication and Concurrency

3.1 Objectives

This lab is to learn about, and gain practical experience in interprocess communication and concurrency control in a general Linux environment. Shared memory allows multiple processes to share a given region of memory. It is the fastest form for different processes to communicate. Processes need to take care of the shared memory conflicting operations. The operating system provides concurrency control facility such as semaphore API.

After this lab, students will be able to

- design and implement a multi-processes concurrent program by using the producer-consumer pattern;
- program with
  - the `fork()` system call to create a new child process;
  - the `wait()` family system calls to obtain the status-change information of a child process;
  - the Linux shared memory API to allow processes to communicate; and
  - the Linux semaphore facility to synchronize processes.

3.2 Starter Files

The starter files for Lab 3 contain the following:

- the fork has example code of creating multiple processes and time the total execution time; it also demonstrate how a zombie process is created when the parent process does not call `wait` family calls;
- the sem has example code of using POSIX semaphore shared between processes;
- the shm has example code of using System V shared memory; and
- the cURL_IPC has example code of using a shared memory region as a cURL call back function buffer to download one image segment from a lab server by the child process and writing the downloaded image segment to a file by the parent process; and
- the tools has a shell script to compute statistics of timing data and a shell script to clean IPC facilities.

The lab3_eceubunt1.csv is the template file that you will need for submitting timing results (see Section 3.5.2).

3.3 Pre-lab Preparation

Read the entire lab3 manual to understand what the lab assignment is about. Build and run the starter code to see what they do. You should work through the provided starter code to understand how they work. The following activities will help you to understand the code.

1. Execute `man fork` to read the man page of `fork(2)`.
2. Execute `man 2 wait` to read the man page of `wait(2)` family system calls.
3. Execute `man ps` to read the man page of the `ps` command.
4. Execute `man shm_overview` to read Linux man page of POSIX shared memory API overview. At the bottom of the man page, it talks about system V shared memory facilities. Read the corresponding man pages of the system V shared memory API.
5. Execute `man sem_overview` to read Linux man page of POSIX semaphore API overview.
6. Execute `man ipcs` and `man ipcrm` to read the man pages of Linux IPC facility commands. You will find the `-s` and the `-m` options are helpful in this lab.

Linux man pages are also available on line at https://linux.die.net/.

The main data structure to represent the fixed size buffer is a queue. A circular queue is one commonly seen implementation of a fixed size buffer if FIFO is required. A stack is another implementation if LIFO is required. You can either create the data structure yourself or use one from an existing library. If you want to explore the C library queue facilities, check out the man pages of insque(3), remque(3) and queue(3). There are example code at the end of the man pages.

3.4 Lab Assignment

3.4.1 The Producer Consumer Problem

A producer-consumer problem is a classic multi-tasking problem. There are one or more tasks that create data and they are referred to as *producers*. There are one or more tasks that use the data and they are referred to as *consumers*. We will have a system of P producers and C consumers. Producers and consumers do not necessarily complete their tasks at the same speed. How many producers should be created and how many consumers should be created to achieve maximum latency improvement[1]? What if the buffer receiving the produced data has a fixed size? Another problem to think about is that when we fix the number of producers and consumers, how big the bounded buffer size should be? Is it true the bigger the buffer size is, the more latency improvement we will get, or there is a limit beyond which the bigger buffer size will not bring any further latency improvement? We will do some experiments to answer these questions by solving a similar problem that we solved in lab2 with some additional assumptions.

In lab2 we used multi-threading to download image segments from the web server and then paste all the segments together. This falls into the unbounded buffer producer consumer problem pattern. We can let producers download the image segments (i.e. creating data) and let consumers extract the image pixel data information (i.e. processing data) for future processing. One easy solution to lab 2 (also commonly seen) is to have one thread that does both the producer and the consumer jobs. This implicitly assumes that the number of producers and consumers are equal. But what if data creation and data processing are running at different speeds[2]? It may take more time to download data than to process data or vice versa. Then having the same number of producers and consumers are not optimal. In addition, in lab2, we did not restrict the receiving data buffer size. In a real world, resources are limited and the situation that a fixed size of buffer space to receive the incoming data is more realistic. In this lab, we have the additional constraint that the buffer to receive the image segments from the web server has a fixed size. So the problem we are solving

[1]You probably have already noticed in lab2 that once the number of threads reaches a certain number, you reach the maximum performance improvement.

[2]For example, the data processing part could be more involved such as doing some image transformation. It could also be that the network bandwidth is tight or the lab server is slow so that it takes long to download the image segment.

is a bounded buffer producer-consumer problem[3].

3.4.2 Problem Statement

We are still solving an image concatenation problem. The image strips are the same ones that you have seen in lab2. In lab2, the lab web server sleeps a random seconds before it sends a random horizontal strip of an image to the client. In this lab, we have a different server running at port 2530 which sleeps for a fixed time before it sends a specific image strip requested by the client. The deterministic sleep time in the server is to simulate the time to produce the data. The image format sent by the server is still the simple PNG format (see Figure 1.2(a)). The PNG segment is still an 8-bit RGBA/color image (see Table 1.1 for details). The web server still uses an HTTP response header that includes the sequence number to tell you which strip it sends to you. The HTTP response header has the format of "X-Ece252-Fragment: M" where $M \in [0, 49]$. To request a horizontal strip with sequence number M of picture N, where $N \in [1, 3]$, use the following URL: `http://machine:2530/image?img=N&part=M`, where

`machine` is one of the following:

- `ece252-1.uwaterloo.ca`,
- `ece252-2.uwaterloo.ca`, and
- `ece252-3.uwaterloo.ca`.

For example, when you request data from http://ece252-1.uwaterloo.ca:2530/image?img=1&part=2, you will receive a horizontal image strip with sequence number 2 of picture 1 . The received HTTP header will contain "X-Ece252-Fragment: 2". The received data will be the image segment in PNG format. You may use the browser to view a horizontal strip of the PNG image the server sends. Each strip has the same dimensions of 400×6 pixels and is in PNG format.

Your objective is to request all horizontal strips of a picture from the server and then concatenate these strips in the order of the image sequence number from top to bottom to restore the original picture as quickly as possible for a given set of given input arguments specified by the user command. You should name the concatenated image as `all.png` and output it to the current working directory.

There are three types of work involved. The first is to download the image segments. The second is to process downloaded image data and copy the processed data to a global data structure for generating the concatenated image. The third is

[3] Here is another producer consumer problem example: you can think of the producer as a keyboard device driver and the consumer as the application wishing to read keystrokes from the keyboard; in such a scenario the person typing at the keyboard may enter more data than the consuming program wants, or conversely, the consuming program may have to wait for the person to type in characters. This is, however, only one of many cases where producer/consumer scenarios occur, so do not get too tied to this particular usage scenario.

to generate the concatenated all.png file once the global data structure that holds the concatenated image data is filled.

The producers will make requests to the lab web server and together they will fetch all 50 distinct image segments. Each time an image segment arrives, it gets placed into a fixed-size buffer of size B, shared with the consumer tasks. When there are B image segments in the buffer, producers stop producing. When all 50 distinct image segments have been downloaded from the server, all producers will terminate. That is the buffer can take maximum B items, where each item is an image segment. The horizontal image strips sent out by the lab servers are all less than $10,000$ bytes.

Each consumer reads image segments out of the buffer, one at a time, and then sleeps for X milliseconds specified by the user in the command line[4]. Then the consumer will process the received data. The main work is to validate the received image segment and then inflate the received IDAT data and copy the inflated data into a proper place inside the memory.

Given that the buffer has a fixed size, B, and assuming that $B < 50$, it is possible for the producers to have produced enough image segments that the buffer is filled before any consumer has read any data. If this happens, the producer is blocked, and must wait till there is at least one free spot in the buffer.

Similarly, it is possible for the consumers to read all of the data from the buffer, and yet more data is expected from the producers. In such a case, the consumer is blocked, and must wait for the producers to deposit one or more additional image segments into the buffer.

Further, if any given producer or consumer is using the buffer, all other consumers and producers must wait, pending that usage being finished. That is, all access to the buffer represents a critical section, and must be protected as such.

The program terminates when it finishes outputting the concatenated image segments in the order of the image segment sequence number to a file named all.png.

Note that there is a subtle but complex issue to solve. Multiple producers are writing to the buffer, thus a mechanism needs to be established to determine whether or not some producer has placed the last image segment into the buffer. Similarly, multiple consumers are reading from the buffer, thus a mechanism needs to be established to determine whether or not some consumer has read out the last image segment from the buffer [5].

[4]This is to simulate data processing takes time.

[5]Due to network transmission has randomness, the order of image segments placed in the buffer may not necessarily be the same order that they have been requested by the producers. The last image segment in the buffer may not necessarily be the image segment with the biggest sequence number. We do not want to request the same image segment twice since this will bring down the performance, so both producers and consumers know the buffer in total will serve 50 image segments.

Requirements

Let B be the buffer size, P be the number of producers, C be the number of consumers, X be the number of milliseconds that a consumer sleeps before it starts to process the image data, and N be the image number you want to get from the server. The producer consumer system is called with the execution command syntax of:

```
./paster2 <B> <P> <C> <X> <N>
```

The command will execute per the above description and will then print out the time it took to execute. You should measure the time before you create the first process and the time after the last image segment is consumed and the concatenated all.png image is generated. Use the `gettimeofday` for time measurement (see starter code under the `fork` directory) and terminal screen for display. Thus your last line of output should look like:

```
paster2 execution time: <time in seconds> seconds
```

For a set of given `(B, P, C, X, N)` tuple values, run your application and measure the time it takes. Note for a give value of `(B, P, C, X, N)`, you need to run multiple times to compute the average execution time in a general Linux environment.

Implement each producer/consumer as an individual process. You start your program with one process which then forks multiple producer processes and multiple consumer processes. The parent process will wait for all the child processes to terminate and then start to process the data structure that holds the concatenated image data and create the final all.png file. Aside from the parent process, the P producer processes that download the image segments and C consumer processes that process the image segment data, you are allowed to create extra processes to do other type of work when you see a need. Just keep in mind that having more processes is not cost free. Hence a good implementation will try to minimize system resource usage unless extra resource usage will bring meaningful improvement.

Use shared memory for processes to communicate. You may use System V shared memory API. The bounded buffer is a shared data structure such as a circular queue that all processes share access to. Note that shared memory access needs to be taken care of at the application level. The POSIX semaphore are to be used for concurrency control.

A Sample Program Run

The following is an example execution of paster2 given $(B, P, C, X, N) = (2, 1, 3, 10, 1)$. In this example, the bounded buffer size is 2. We have one producer to download the image segments and three consumers to process the downloaded data. Each consumer sleeps 10 milliseconds before it starts to process the data. And the image segments requested are from image 1 on lab servers.

```
[eceubuntu1:]./paster2 2 1 3 10 1
paster2 execution time: 100.45 seconds
```

Note that due to concurrency, your output may not be exactly the same as the sample output above. Also depending on the implementation details and the platform where the program runs, the sample system execution time is only for illustration purpose. The exact paster2 execution time value your program produces will be different than the one shown in the sample run.

3.5 Deliverables

3.5.1 Pre-lab Deliverables

None.

3.5.2 Post-lab Deliverables

Put the following items under the directory named lab3:

1. All the source code and a Makefile. The Makefile default target is `paster2` executable file. That is command `make` should generate the `paster2` executable file. We also expect that the command `make clean` will remove the object code and the default target. That is the `.o` files and the executable files should be removed.

2. A timing result .csv file named lab3_hostname.csv which contains the timing results by running paster2 on a server whose name is hostname. For example, `lab3_eceubuntu1.csv` means paster2 was executed on the server eceubuntu1 and the file contains the timing results. The first line of the file is the header of the timing result table. The rest of the rows are the timing result command line argument values and the timing results. The columns of the .csv file from left to right are values of B, P, C, X, and the corresponding paster2 average execution time. We have an example .csv file in the starter code folder named `lab3_eceubuntu1.csv` for illustration purpose.
   Run your paster2 on `eceubuntu1`. Record the average timing measurement data for the (B, P, C, X, N) values shown in Table 3.1 for a particular host. Note that for each given (B, P, C, X, N) value in the table, you need to run the program n times and compute the average time. We recommend $n = 5$.
   The lab is submitted by pushing to the group's gitlab repository before the deadline.

- Put the entire source code with a Makefile under the directory lab3. The Makefile default target is `paster2`. That is command `make` should generate the aforementioned executable file. We also expect that command `make clean` will remove the object code and the default target. That is the `.o` files and the executable file should be removed.
- Use `git add` to add the files to the git repo.
- Use `git commit` to commit the changes. The commit message should be "Lab 3 Submission". The final such commit will be considered.
- Use `git push` to push the changes to your repo, thereby performing the actual submission. Without pushing, a submission is not made.

B	P	C	X	N	Time
5	1	1	0	1	
5	1	5	0	1	
5	5	1	0	1	
5	5	5	0	1	
10	1	1	0	1	
10	1	5	0	1	
10	1	10	0	1	
10	5	1	0	1	
10	5	5	0	1	
10	5	10	0	1	
10	10	1	0	1	
10	10	5	0	1	
10	10	10	0	1	
5	1	1	200	1	
5	1	5	200	1	
5	5	1	200	1	
5	5	5	200	1	
10	1	1	200	1	
10	1	5	200	1	
10	1	10	200	1	
10	5	1	200	1	
10	5	5	200	1	
10	5	10	200	1	
10	10	1	200	1	
10	10	5	200	1	
10	10	10	200	1	
5	1	1	400	1	
5	1	5	400	1	
5	5	1	400	1	
5	5	5	400	1	
10	1	1	400	1	
10	1	5	400	1	
10	1	10	400	1	
10	5	1	400	1	
10	5	5	400	1	
10	5	10	400	1	
10	10	1	400	1	
10	10	5	400	1	
10	10	10	400	1	

Table 3.1: Timing measurement data table for given (B, P, C, X, N) values.

3.6 Marking Rubric

The Rubric for marking is listed in Table 3.2.

Points	Sub-points	Description
100		Post-lab
	10	clean project files organization
	5	`Makefile`
	85	Complete implementation of `paster2` and timing data

Table 3.2: Lab3 Marking Rubric

Lab 4

A Multi-threaded Web Crawler

4.1 Objectives

This lab is to design and implement a multi-threaded web crawler. In the previous lab, we practised memory sharing between processes as a means to communicate between processes. Sharing memory between threads are a lot easier since they live in the same address space. We do not need the operating system's involvement to have a shared memory region between threads. In addition, creating/destroying threads is less expensive than creating/terminating child processes.

However we still need to avoid race conditions in the memory region that threads are sharing. Aside from mutex and semaphore, the operating system also provides condition variable and atomic type facilities.

After this lab, students will be able to

- design and implement a multi-threaded concurrent program that requires more than one synchronization pattern; and
- gain more experiences in the Linux mutex, semaphore, condition variable and atomic type facilities to synchronize threads.

4.2 Starter Files

The starter files for Lab 4 contain the following:

- the curl_xml has example code to show how to use curl and libxml2 together to identify a possible png page and extract http(s) links from a html page.
- the tools has a shell script to compute statistics of timing data.

The lab4_eceubunt1.csv is the template file that you will need for submitting timing results (see Section 4.5.2).

4.3 Pre-lab Preparation

Read the entire lab4 manual to understand what the lab assignment is about. Build and run the starter code to see what they do. You should work through the provided starter code to understand how they work. The following activities will help you to understand the code.

1. Run the given starter code with the following URLs and examine responses from the server in the http header.
   - http://ece252-1.uwaterloo.ca/lab4
   - http://ece252-1.uwaterloo.ca/lab3/index.html
   - http://ece252-1.uwaterloo.ca/~yqhuang/lab4/Disguise.png
   - http://ece252-1.uwaterloo.ca:2530/image?img=1&part=1
2. Execute `man pthread_cond` to read the man page of condition variable.

Linux man pages are also available on line at https://linux.die.net/.

4.4 Lab Assignment

4.4.1 Problem Statement

In the previous labs, the URLs[1] to download the image segments are given. In this lab, you will need to search some HTTP lab servers to find these URLs. We have 50 different URLs, each of which links to a unique PNG image segment of a particular image. The mission is to search for these URLs on the lab servers[2].

To solve the problem, we will create a multi-threaded web crawler named `findpng2` to search the web given a seed URL and find all the URLs that link to PNG images.

[1] URL stands for Uniform Resource Locator (see https://en.wikipedia.org/wiki/URL). It is a web page address. For the purpose of this lab, it starts with the string "http://"

[2] This lab does not require one to concatenate these segments. However if you are interested in what these segments are, then you can use your catpng to restore the original image after downloading all the segments or directly concatenate the segments in memory using lab2/3 code. The simple PNG format, dimensions of each image segment and the http header that tells you which segment you are getting are the same as what we had in previous labs.

4.4.2 The findpng2 command

The expected behaviour of the `findpng2` is given in the following manual page of the command.

Man page of findpng2

NAME

findpng2 - search for PNG file URLs on the web

SYNOPSIS

findpng2 [OPTION]... SEED_URL

DESCRIPTION

Start from the SEED_URL and search for PNG file URLs on the world wide web and return the search results to a plain text file named `png_urls.txt` in the current working directory. Output the execution time in seconds to the standard output.

-t=NUM

create NUM threads simultaneously crawling the web. Each thread uses the curl blocking I/O to download the data and then process the downloaded data. The total number of pthread_create() invocations should equal to NUM specified by the -t option. When this option is not specified, assumes a single-threaded implementation.

-m=NUM

find up to NUM of unique PNG URLs on the web. It is possible that the search results is less than NUM of URLs. When this option is not specified, assumes NUM=50.

-v=LOGFILE

log all the visited URLs by the crawler, one URL per line in LOGFILE. When this option is not specified, do not log any visited URLs by the crawler and do not create any visited URLs log file.

OUTPUT FORMAT

The time to execute the program is output to the standard output. It will look like the following:

```
findpng2 execution time: S seconds
```

The search results is a list of PNG URLs, one URL per line saved in a file named `png_urls.txt` . The order of listing the search results is not specified. If the search result is empty, then create an empty search result file.

EXAMPLES

```
findpng2 -t 10 -m 20 -v log.txt http://ece252-1.uwaterloo.ca/lab4
```

Find up to 20 PNG URLs starting from http://ece252-1.uwaterloo.ca/lab4 using 10 threads. The output on the standard output will look like the following:

```
findpng2 execution time: 10.123456 seconds
```

The first two lines in the `png_urls.txt` file may look like the following:

```
http://ece252-2.uwaterloo.ca:2540/img?q=tyfoighidfyseoid==
http://ece252-1.uwaterloo.ca:2541/img?q=kjvjkjxsroutqpqkgh
```

An empty search result will generate `an empty png_urls.txt` file.

The first two lines in the `log.txt` file may look like the following:

```
http://ece252-1.uwaterloo.ca/lab4
http://ece252-1.uwaterloo.ca/~yqhuang/lab4/index.html
```

4.4.3 Web crawling

The `findpng2` is a tiny simplified web crawler. It searches the web by starting from a seed URL. The crawler visits the given URL page and finds two pieces of information.

The first piece is the URLs that link to valid PNG images[3]. The crawler adds PNG URLs found to a search result table. We want this table to contain unique URLs, hence if the found URL is already in the table, you should not add it to the table.

The second piece is a set of new URLs to further crawl. The crawler adds this set of new URLs to a URLs pool known as the `URLs frontier` . Since visiting web pages has costs, we do not want the crawler to visit the same page twice. Hence the crawler needs a mechanism to remember URLs that have been visited already. As the crawler visits the URLs in the URLs frontier, the process of finding the target PNG URLs and new URLs to further explore repeats until it finds no more new PNG URL or it reaches the maximum number of PNG URLs specified by the user input.

[3] A valid PNG image is a file whose first 8 bytes matches the PNG signature bytes

4.4.4 The HTTP

HTTP stands for "Hypertext Transfer Protocol". They carry important information about the client requests and the server responses. When the client sends an URL to the server, it makes an HTTP GET request to the server and the detailed information about the request is in the headers. The server will first respond with an HTTP response status code line. There are three categories we need to handle in this lab.

- HTTP/1.1 2XX. This is a success response. We need to process the data the link gives.
- HTTP/1.1 3XX. This is the case the link has been relocated. By feeding the `curl_easy_setopt` with the `CURLOPT_FOLLOWLOCATION`, curl will follow the relocated links. The `CURLOPT_MAXREDIRS` in the curl option setting allows one to specify maximum number of redirects to follow.
- HTTP/1.1 4XX. This is a broken link, usually caused by the client side. We do not process the link. But we need to remember this link has been visited.
- HTTP/1.1 5XX. This is also a broken link, usually caused by server internal error. We are not able to process the link. But again need to remember this link has been visited.

After the response status code line, the web server uses http response headers to send meta information in different fields about the web resource content it sends to the client. One of the fields is "Content-Type". For the purpose of the lab, we are only interested in two types of Content-Type. One is the `text/html`, which is a hyper text file where we find more URLs. The other one is the `image/png` which is a PNG image that we look for. You will process the following two cases of Content-Type:

- Content-Type: text/html
- Content-Type: image/png

The http header call back function of curl allows us to process all the header responses from the server. Another way is to use `curl_easy_getinfo` function to obtain a specific header information[4]. For example, with the second parameter of the function setting to `CURLINFO_CONTENT_TYPE`, we obtain the content type header information.

As you may recall from previous lab, the lab server uses an HTTP response header that has the format of "X-Ece252-Fragment: M" where $M \in [0, 49]$ to tell which image segment it sends to the client. If you are only interested in finding the PNG image segments that the lab web server has, then this piece of information is useful.

[4] Only standardized headers are supported. User defined headers such as those starting with X- are not supported.

After all the response headers are sent, the server sends out the actual contents of the web resource in the message body. The write call back function of curl allows us to process this piece of information.

4.4.5 Programming Tips

You will need a number of lists to keep track of different sets of URLs. One list is for the URLs frontier, which contains to-be-visited URLs. One list is for recording all the URLs that have been visited. Another list is for the PNG URLs that have been found. To crawl the web using multiple threads, these lists are shared between threads. Hence you need to synchronize them. Some lists can only be accessed by one thread both when reading and writing. Some lists may be accessed by multiple threads when reading, but only one thread when writing.

Another subtle difficulty is to know when to terminate the program. The program should terminate either when there are no more URLs in the URLs frontier or the user specified number of PNG URLs have been found. You may need some shared counters to keep track of information such as how many PNG URLs have been found and how many threads are waiting for a new URL.

If an URL has been visited already, then we do not want to visit it again. So a search of visited-URLs list is needed. Hashing will make the search very effective and you may consider using a hash table to represent this already-visited list. The glibc has hash table API (`man hsearch(3)`).

4.5 Deliverables

4.5.1 Pre-lab deliverables

None.

4.5.2 Post-lab Deliverables

Put the following items under the directory named lab4:

1. All the source code and a Makefile. The Makefile default target is `findpng2` executable file. That is command `make` should generate the `findpng2` executable file. We also expect that the command `make clean` will remove the object code and the default target. That is the `.o` files and the executable file should be removed.

2. A timing result .csv file named lab4_hostname.csv which contains the timing results by running the findpng2 on a server whose name is hostname. For example, `lab4_eceubuntu1.csv` means findpng2 was executed on the server

eceubuntu1 and the file contains the timing results. The first line of the file is the header of the timing result table. The rest of the rows are the timing result command line argument values and the timing results. The columns of the .csv file from left to right are values of T (the number of threads), M (the number of unique PNG links to search for) and $TIME$ (the corresponding findpng2 average execution time). We have an example .csv file in the starter code folder named `lab4_eceubuntu1.csv` for illustration purpose.

Run your findpng2 on eceubuntu1. Record the average timing measurement data for the (T, M) values shown in Table 4.1 for a particular host. Note that for each given (T, M) value in the table, you need to run the program n times and compute the average time. We recommend $n = 5$.

T	M	Time
1	1	
1	10	
1	20	
1	30	
1	40	
1	50	
1	100	
10	1	
10	10	
10	20	
10	30	
10	40	
10	50	
10	100	
20	1	
20	10	
20	20	
20	30	
20	40	
20	50	
20	100	

Table 4.1: Timing measurement data table for given (T, M) values.

The lab is submitted by pushing to the group's gitlab repository before the deadline.

- Use `git add` to add the files to the git repo.
- Use `git commit` to commit the changes. The commit message should be "Lab 4 Submission". The final such commit will be considered.

- Use `git push` to push the changes to your repo, thereby performing the actual submission. Without pushing, a submission is not made.

4.6 Marking Rubric

Table 4.2 shows the rubric for marking the lab.

Points	Sub-points	Description
100		Post-lab
	10	clean project files organization
	5	Makefile correctly builds and cleans
	65	Implementation of multi-threaded `findpng2`
	20	Correct timing results in lab4_hostname.csv file

Table 4.2: Lab4 Marking Rubric

Lab 5

Asynchronous I/O with cURL

5.1 Objectives

This lab is to design and implement a single-threaded web crawler. In the previous lab, we designed and implemented a multi-threaded concurrent web crawler by using blocking I/O with cURL in each thread. Another solution to make the program concurrent is to use non-blocking I/O known as asynchronous I/O. The cURL multi interface enables multiple simultaneous transfers in the same thread.

After this lab, students will be able to

- design and implement a single-threaded concurrent program by using asynchronous I/O; and
- gain experiences in cURL multi-interface.

5.2 Starter Files

The starter files for Lab 5 contain the following:

- the curl_multi has example code to show how to use curl multi interface API; and
- the tools has a shell script to compute statistics of timing data.

The lab5_eceubunt1.csv is the template file that you will need for submitting timing results (see Section 5.5.2).

5.3 Pre-lab Preparation

Build and run the starter code to see what they do. You should work through the provided starter code to understand how they work. Read the documentation of curl multi interface at the following URLs:

- The curl multi interface overview; and
- Driving with multi interface.

5.4 Lab Assignment

5.4.1 Problem Statement

We are still solving the PNG URLs searching problem defined in lab4. Instead of creating a multi-threaded web crawler, we will create a single-threaded concurrent web crawler by using non-blocking I/O to enable simultaneous transfers. You will need to use the curl multi API.

This time, the solution should *not* use pthreads. However, it should keep multiple concurrent connections to servers open. We will create a single-threaded web crawler named `findpng3` to search the web given a seed URL and find all the URLs that link to PNG images.

5.4.2 The findpng3 command

The expected behaviour of the `findpng3` is given in the following manual page of the command.

Man page of findpng3

NAME

> **findpng3** - search for PNG file URLs on the web using a single thread

SYNOPSIS

> **findpng3** [OPTION]... SEED_URL

DESCRIPTION

> Start from the SEED_URL and search for PNG file URLs on the world wide web and return the search results to a plain text file named `png_urls.txt`

in the current working directory. Output the execution time in seconds to the standard output.

-t=NUM

keep maximum NUM concurrent connections[1] to servers open when crawling the web. When this option is not specified, assumes a single connection.

-m=NUM

find up to NUM of unique PNG URLs on the web. It is possible that the search results is less than NUM of URLs. When this option is not specified, assumes NUM=50.

-v=LOGFILE

log the visited URLs by the crawler, one URL per line in LOGFILE.

OUTPUT FORMAT

The time to execute the program is output to the standard output. It will look like the following:

```
findpng3 execution time: S seconds
```

The search results is a list of PNG URLs, one URL per line saved in a file named `png_urls.txt` . The order of listing the search results is not specified. If the search result is empty, then create an empty search result file.

EXAMPLES

```
findpng3 -t 10 -m 20 -v log.txt http://ece252-1.uwaterloo.ca/lab4
```

Find up to 20 PNG URLs starting from http://ece252-1.uwaterloo.ca/lab4 by keeping maximum 10 concurrent connections open to servers. The output on the standard output will look like the following:

```
findpng3 execution time: 10.123456 seconds
```

The first two lines in the `png_urls.txt` file may look like the following:

```
http://ece252-2.uwaterloo.ca:2540/img?q=tyfoighidfyseoid==
http://ece252-1.uwaterloo.ca:2541/img?q=kjvjkjxsroutqpqkgh
```

[1]There are two implementation options when you launch up to NUM transfer requests. One is to launch up to NUM transfer requests in batch, wait all of them to complete and then move to the next group of requests in batch. Another one is immediate replacement of the individual handle. We recommend the second approach. But the first one is accepted.

An empty search result will generate `an empty png_urls.txt` file.

The first two lines in the `log.txt` file may look like the following:

```
http://ece252-1.uwaterloo.ca/lab4
http://ece252-1.uwaterloo.ca/~yqhuang/lab4/index.html
```

5.5 Deliverables

5.5.1 Pre-lab Deliverables

None.

5.5.2 Post-lab Deliverables

Put the following items under the directory named lab5:

1. All the source code and a Makefile. The Makefile default target is `findpng3` executable file. That is command `make` should generate the `findpng3` executable file. We also expect that the command `make clean` will remove the object code and the default target. That is the `.o` files and the executable file should be removed.

2. A timing result .csv file named lab5_hostname.csv which contains the timing results by running the findpng3 on a server whose name is hostname. For example, `lab5_eceubuntu1.csv` means findpng3 was executed on the server eceubuntu1 and the file contains the timing results. The first line of the file is the header of the timing result table. The rest of the rows are the timing result command line argument values and the timing results. The columns of the .csv file from left to right are values of T (the number of threads), M (the number of unique PNG links to search for) and $TIME$ (the corresponding findpng3 average execution time). We have an example .csv file in the starter code folder named `lab5_eceubuntu1.csv` for illustration purpose.

   Run your findpng3 on eceubuntu1. Record the average timing measurement data for the (T, M) values shown in Table 5.1 for a particular host. Note that for each given (T, M) value in the table, you need to run the program n times and compute the average time. We recommend $n = 5$.

The lab is submitted by pushing to the group's gitlab repository before the deadline.

- Use `git add` to add the files to the git repo.

- Use `git commit` to commit the changes. The commit message should be "Lab 5 Submission". The final such commit will be considered.

T	M	Time
1	1	
1	10	
1	20	
1	30	
1	40	
1	50	
1	100	
10	1	
10	10	
10	20	
10	30	
10	40	
10	50	
10	100	
20	1	
20	10	
20	20	
20	30	
20	40	
20	50	
20	100	

Table 5.1: Timing measurement data table for given (T, M) values.

- Use `git push` to push the changes to your repo, thereby performing the actual submission. Without pushing, a submission is not made.

5.6 Marking Rubric

The Rubric for marking is listed in Table 5.2.

Points	Description
10	clean project files organization
5	Makefile correctly builds and cleans `findpng3`
65	Implementation of `findpng3` in Section 5.4.2
20	Correct timing results in lab5_hostname.csv file

Table 5.2: Lab5 Marking Rubric

Part III

Software Development Environment Quick Reference Guide

Chapter 1

Introduction to ECE Linux Programming Environment

1.1 ECE Linux Servers

There are a group of Linux Ubuntu servers that are open to ECE undergraduate students. The machines are listed at url: https://ece.uwaterloo.ca/Nexus/arbeau/clients. To access one of the machines, we recommend to use the alias name of `eceubuntu.uwaterloo.ca`, which will direct the user to the most lightly loaded machine at the time of login.

To access these machines from off campus. One way is to use the campus VPN. Another way is to first connect to `eceterm.uwaterloo.ca` and then connect to other Linux servers from there. Note that the `eceterm` should not be used for computing jobs, it is for accessing other Linux servers on campus.

1.2 Connecting to Linux servers

A terminal client software that supports secure shell (ssh) will allow you to remotely connect to the Linux servers. MobaXterm is a convenient application that not only supports ssh, but also has a built-in X server that allows one to run Graphical User Interface (GUI) applications from the Linux servers.

Use the File Explorer to navigate to `Q:\eng\ece\Util` folder, scroll down until you find the MobaXterm icon and double click it (see Figure C1). The MobaXterm window will pop up. There is a grey rectangular button labelled “Start local terminal” in the middle (see Figure C2). Click this button.

Then a terminal session starts. You will need to use the command line `ssh` command to connect to the Linux server. Use your UWID and password to login. The syntax of the command is as follows:

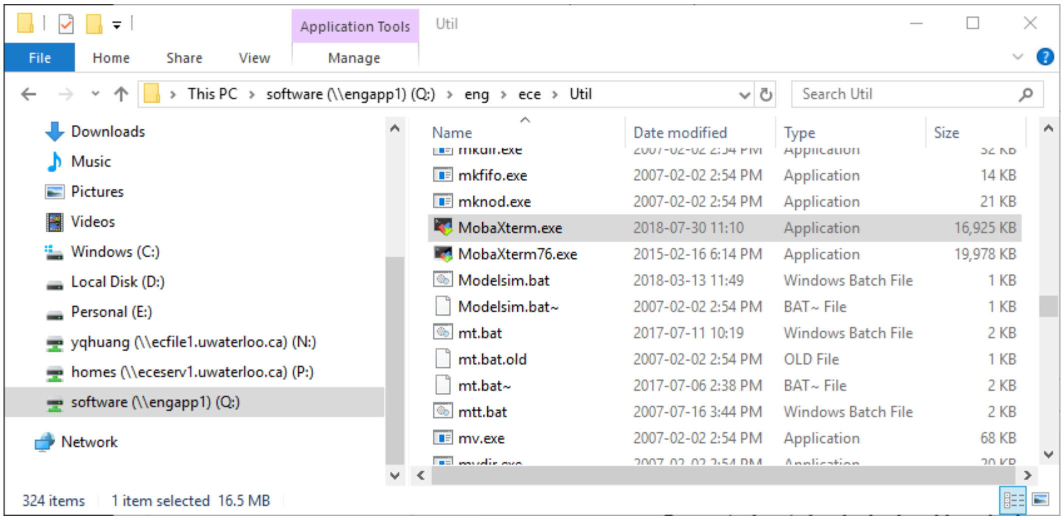


Figure C1: MobaXterm Path on ECE Nexus Windows 10 Machines.

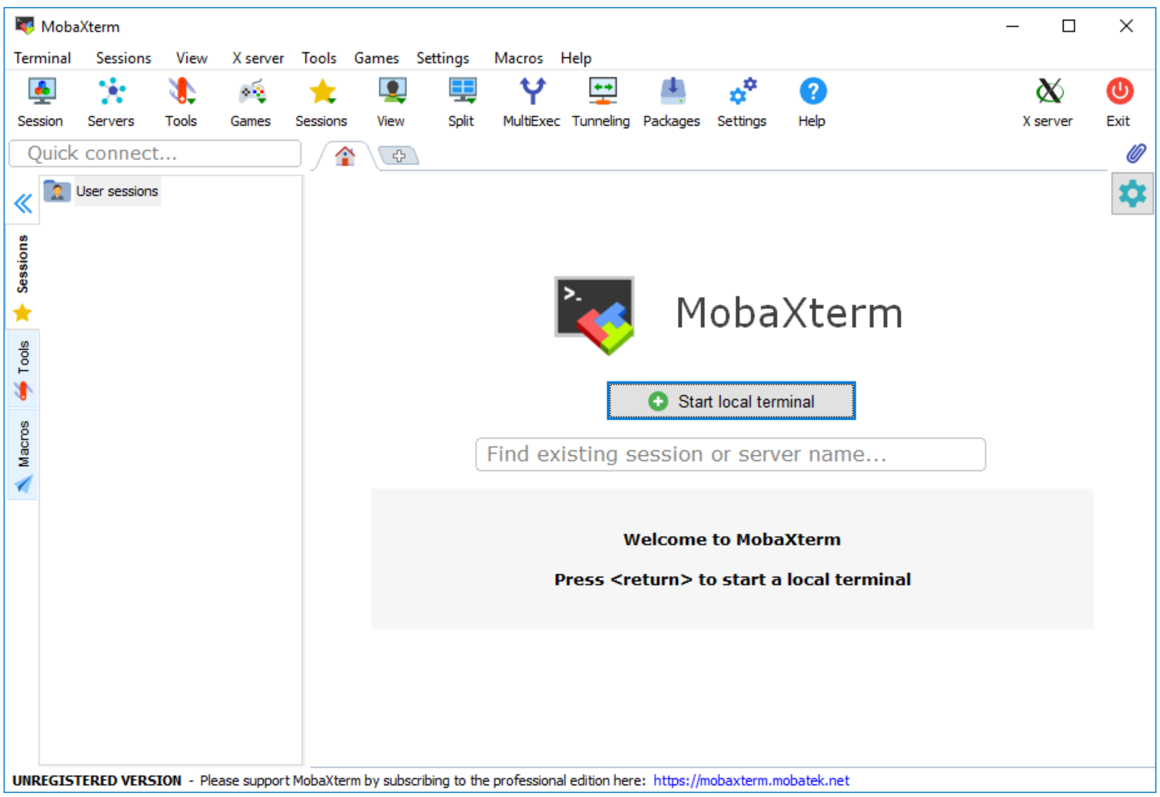


Figure C2: MobaXterm Welcome Page.

```
ssh -XY <UWID>@eceubuntu.uwaterloo.ca
```

See Figure C3 for reference.

All your ECE Linux account files are accessible through P Drive on Nexus ma-

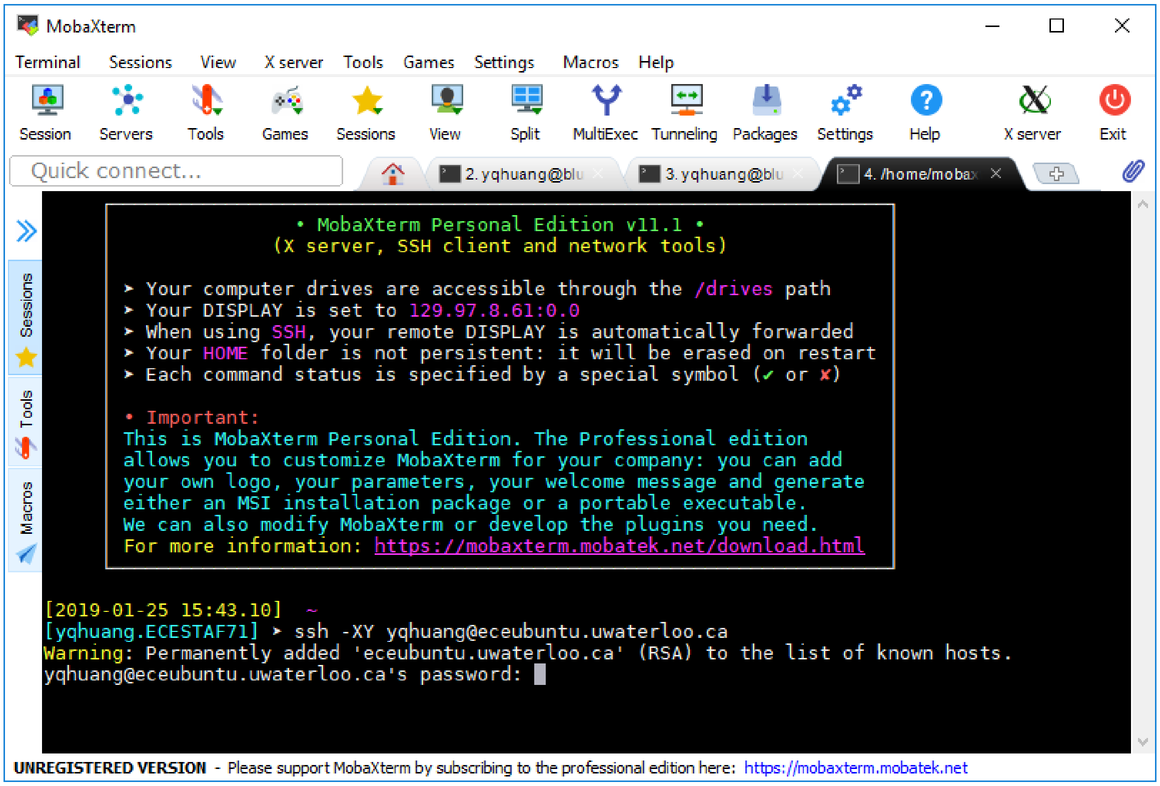


Figure C3: MobaXterm Welcome Page.

chines (See Figure C4). The P Drive is only accessible within campus network. Mapping the Linux account as a network drive off campus is not supported due to security reasons.

1.3 Basic Software Development Tools

To develop a program, there are three important steps. First, a program is started from source code written by programmers. Second, the source code is then compiled into object code, which is a binary. Non-trivial project normally contains more than one source file. Each source file is compiled into one object code and the linker would finally link all the object code to generate the final target, which is the executable that runs. The steps of compiling and linking are also known as building a target. It is very rare that the target will run perfectly the first time it is built. Most of time we need to fix defects and bugs in the code and the this is the third step. The debugger is a tool to help you identify the bug and fix it. Table C1 shows the key steps in programming work flow and example tools provided by a general purpose Linux operating system.

Most of you probably are more familiar with a certain Integrated Development Environment (IDE) which integrates all these tools into a single environment. For

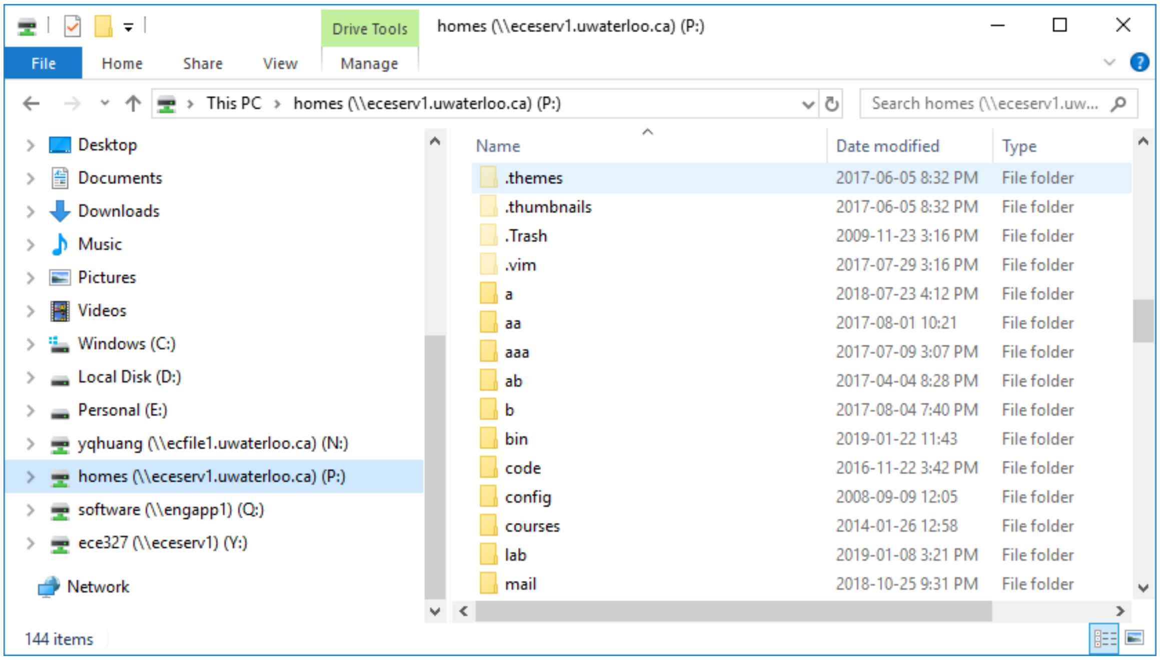


Figure C4: Linux files on P drive, a network mapped drive.

Task	Tool	Examples
Editing the source code	Editor	vi, emacs
Compiling the source code	Compiler	gcc
Debugging the program	Debugger	gdb, ddd

Table C1: Programming Steps and Tools

example Eclipse and Visual Studio. A different approach is to select a tool in each programming step and build your own tool chain. Many seasoned Linux programmers build their own tool chains. A few popular tools are introduced in the following subsections.

1.3.1 Editor

Some editors are designed to better suit programmers' needs than others. The *vi* (*vim* and *gvim* belong to the vi family) and *emacs* (*xemacs* belongs to emacs family) are the two most popular editors for programming purposes.

Two simple notepad editors *pico* and *nano* are also available for a simple editing job. These editors are not designed for programming activity. To use one of them to write your first *Hello World* program is fine though.

After you finish editing the C source code, give the file name an extension of .c. Listing 1.1 is the source code of printing "Hello World!" to the screen.

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  printf("Hello World!\n");
  exit(0);
}
```

Listing 1.1: HelloWorld C source Code

1.3.2 C Compiler

The source code then gets fed into a compiler to be come an executable program. The GNU project C and C++ compiler is *gcc*. To compile the HelloWorld source code in Listing 1.1, type the following command at the prompt:

```
gcc helloworld.c+
```

You will notice that a new file named `a.out` is generated. This is the executable generated from the source code. To run it, type the following command at the prompt and hit Enter.

```
./a.out
```

The result is"Hello World!" appearing on the screen.

You can also instruct the compiler to name the executable another name instead of the default `a.out`. The `-o` option in gcc allows one to name the executable a name. For example, the following command will generate an executable named "`helloworld.out`".

```
gcc helloworld.c -o helloworld.out
```

although there is no requirement that the name ends in `.out`.

1.3.3 Debugger

The GNU debugger gdb is a command line debugger. Many GUI debugger uses gdb as the back-end engine. One GNU GUI debugger is *ddd*. It has a powerful data display functionality.

GDB needs to read debugging information from the binary in order to be able to help one to debug the code. The `-g` option in gcc tells the compiler to produce such debugging information in the generated executable. In order to use gdb to debug our simple HelloWorld program, we need to compile it with the following command:

```
gcc -g helloworld.c -o helloworld.out
```

The following command calls gdb to debug the helloworld.out

```
gdb helloworld.out
```

This starts a gdb session. At the `(gdb)` prompt, you can issue gdb command such as `b main` to set up a break point at the entry point of main function. The `l` lists source code. The `n` steps to the next statement in the same function. The `s` steps into a function. The `p` prints a variable value provided you supply the name of the variable. Type `h` to see more gdb commands.

Compared to gdb command line interface, the ddd GUI interface is more user friendly and easy to use. To start a ddd session, type the command

```
ddd
```

and click File → Open Program to open an executable such as helloworld.out. You will then see gdb console in the bottom window with the source window on top of the gdb console window. You could see the value of variables of the program through the data window, which is on top of the source code window. Select View to toggle all these three windows.

A good gdb/ddd tutorial URL is https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_gdb.php. Pay special attention to the section on how to trouble shoot segmentation fault problems.

1.4 More on Development Tools

For any non-trivial software project, it normally contains multiple source code files. Developers need tools to manage the project build process. Also project normally are done by several developers. A version control tool is also needed.

1.4.1 How to Automate Build

Make is an utility to automate the build process. Compilation is a cpu-intensive job and one only wants to re-compile the file that has been changed when you build a target instead of re-compile all source file regardless. The `make` utility uses a Makefile to specify the dependency of object files and automatically recompile files that has been modified after the last target is built.

In a Makefile, one specifies the targets to be built, what prerequisites the target depends on and what commands are used to build the target given these prerequisites. These are the *rules* contained in Makefile. The Makefile has its own syntax. The general form of a Makefile rule is:

```
target ...: prerequisites ...
	recipe
	...
	...
```

One important note is that each recipe line starts with a TAB key rather than white spaces. To build a target, use command `make` followed by the target name or omit the target name to default to the first target in the Makefile. For example

```
make
```

will build your lab starter code.

Listing 1.2 is our first attempt to write a very simple Makefile.

```
helloworld.out: helloworld.c
	gcc -o helloworld.out helloworld.c
```

Listing 1.2: Hello World Makefile: First Attempt

The following command will generate the `helloworld.out` executable file.

```
make helloworld.out
```

Our second attempt is to break the single line gcc command into two steps. First is to *compile* the source code into object code .o file. Second is to *link* the object code to one final executable binary. Listing 1.3 is our second attempted version of Makefile.

```
helloworld.out: helloworld.o
	gcc -o helloworld.out helloworld.o
helloworld.o: helloworld.c
	gcc -c helloworld.c
```

Listing 1.3: Hello World Makefile: Second Attempt

When a project contains multiple files, separating object code compilation and linking stages would give a clear dependency relationship among code. Assume that we now need to build a project that contains two source files `src1.c` and `src2.c` and we want the final executable to be named as `app.out`. Listing 1.4 is a typical example Makefile that is closer to what you will see in the real world.

```
all: app.out
app.out: src1.o src2.o
	gcc -o app.out src1.o src2.o
src1.o: src1.c
	gcc -c src1.c
src2.o: src2.c
	gcc -c src2.c
clean:
	rm *.o app.out
```

Listing 1.4: A More Real Makefile: First Attempt

We also have added a target named *clean* so that `make clean` will clean the build.

So far we have seen the Makefile contains *explicit rules*. Makefile can also contain *implicit rules*, *variable definitions*, *directives* and *comments*. Listing 1.5 is a Makefile that is used in the real world.

```
1 # Makefile to build app.out
2 CC=gcc
3 CFLAGS=-Wall -g
4 LD=gcc
5 LDFLAGS=-g

7 OBJS=src1.o src2.o

9 all: app.out
10 app.out: $(OBJS)
11 	$(LD) $(CFLAGS) $(LDFLAGS) -o $@ $(OBJS)
12 .c.o:
13 	$(CC) $(CFLAGS) -c $<
14 .PHONY: clean
15 clean:
16 	rm -f *.o *.out
```

Listing 1.5: A Real World Makefile

Line 1 is a comment. Lines $2-7$ are variable definitions. Line 12 is an implicit rule to generate .o file for each .c file. See http://www.gnu.org/software/make/manual/make.html to explore more of makefile.

1.4.2 Version Control Software

We use Git version control software. It is installed both on the Linux servers and Nexus windows machines. If you decide to use GitHub to host your repository, please make sure it is a *private* one. Go to http://github.com/edu to see how to obtain five private repositories for two years on GitHub for free.

1.4.3 Integrated Development Environment

Visual Studio Code has been installed on all ECE ubuntu servers. Use the following command to invoke it. Note you will need X Windows support for the GUI interface.

```
code
```

1.5 Man Page

Linux provides manual pages. You can use the command `man` followed by the specific command or function you are interested in to obtain detailed information.

Mange pages are grouped into sections. We list frequently used sections here:

- Section 1 contains user commands.
- Section 2 contains system calls
- Section 3 contains library functions
- Section 7 covers conventions and miscellany.

To specify which section you want to see, provide the section number after the `man` command. For example,

```
man 2 stat
```

shows the system call `stat` man page. If you omit the `2` in the command, then it will return the command `stat` man page.

You can also use `man -k` or `apropos` followed by a string to obtain a list of man pages that contain the string. The Whatis database is searched and now run `man whatis` to see more details of `whatis`.

Appendix A

Forms

Lab administration related forms are given in this appendix.

ECE 252 Request to Leave a Lab Project Group Form

Name:	
Quest ID:	
Student ID:	
Lab Assignment ID	
Group ID:	
Name of Other Group Members:	

Provide the reason for leaving the project group here:

Signature ____________________ Date ____________________

Bibliography

[1] Greg Roelofs. *PNG: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999.