Systems Programming and Concurrency

ECE.252 Lab Manual

Fall 2021

by

Yiqing Huang
Jeff Zarnett
Bernie Roehl
Ricardo Rolon

Electrical and Computer Engineering Department
University of Waterloo

Waterloo, Ontario, Canada, September 9, 2021

Contents

List of Tables

List of Figures

Preface

Who Should Read This Lab Manual?

This lab manual is written for students who are taking the 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. The second purpose of this document is to provide a quick reference guide to the relevant development tools for completing laboratory projects. This manual is divided into three parts:

Part I describes the lab administration policies

Part II describes a set of course laboratory projects as follows:

- Lab1: Systems programming in the 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 to the Linux software development tools. We will be using the Ubuntu 18.04 LTS operating system. The material in this part needs to be self-studied before the labs start. The main topics are as follows.

- The Linux hardware environment
- Editors
- Compiler

- Debugger
- Automated builds
- 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 the ECE.254 and ECE.459 courses in the past few years. They provided constructive feedback every term to make the manual more useful, in order to address problems that students would encounter when working on each lab assignment.

Part I

Lab Administration

Lab Administration Policy

Group Lab Policy

- **Group Size.** All labs are done in groups of *two*. A size of three is only considered in an odd number of students from the course (with the approval of the lab instructor). All group of three requests are processed on a first-come, first-served basis. It is recommended to do the labs in two-member groups due the work amount each lab contains.

  If student is very proficient in C, and with some extra time to invest in this course, it is possible to do it alone. There is no workload reduction if you do the labs individually. Everyone in the group normally gets the same mark. The Learn system (`http://learn.uwaterloo.ca`) is used to sign up for groups. *The lab group signup is due by 24:00 EST on the Second Friday of the academic term*. Late group sign-up is not accepted and will result in losing the entire lab sign-up mark, which is `2%` of the total lab grade. Grace days do not apply to the Group Signup.

- **Group Split-up.** If you notice a workload imbalance in your group, try to solve it as soon as possible within your group. As a last resort, you can split up your group. Group split-up is only allowed once. You are allowed to join a one member group after the split-up. You are not allowed to split up the newly formed group.

  A copy of the code and documentation that was completed before the group split-up will be given to each individual in the group. We highly recommend that everyone stays with their groups as much as possible, since the ability to do work as a team will be an important skill in your future career. Please choose your lab partners carefully.

- **Group Split-up Deadline.** To split up your group starting from a particular lab, you need to notify the lab instructor in writing and sign the group split-up form (see Appendix) a week before the Lab*n* due date. If you are late to submit the split-up form, then you need to finish Lab*n* as a group.

Deliverable	Weight	Lab Due	Time
Group Sign-up	2%	September 17, 2021	24:00 EST
LAB 1	18%	September 24, 2021	24:00 EST
LAB 2	20%	October 08, 2021	24:00 EST
LAB 3	20%	October 29, 2021	24:00 EST
LAB 4	20%	November 12, 2021	24:00 EST
LAB 5	20%	November 26, 2021	24:00 EST

Table 0.1: Project Deliverable Weight of the Lab Grade, Scheduled Lab Sessions and Deadlines.

Lab Assignments Grading and Deadline Policy

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

- **Do not submit anything for Lab 0.** This lab provides basic information about getting connected to eceubuntu Linux server with your userid and password. For more information, view the **ECE 252 Lab 0** video tutorial in Learn system.
- **Lab Assignment Due Dates.** The detailed deadlines of lab deliverables are displayed in Table 0.1. Note that the reading/study week is not counted as a week in the table.
- **Lab Assignment Late Submissions.** Late submission is accepted within five days after the deadline of the lab. No late submission is accepted five days after the lab deadline. There are five grace days [1] that can be used for lab deliverables late submissions.

  After all grace days are consumed, a 10% per day late submission penalty will be applied. However, if it is five days after the lab deadline, no submission is accepted.
- **Lab Re-grading.** To initiate a re-grading process, contact the grading TA in charge first. The re-grading is a rigid process. The entire lab will be re-graded. Your new grades may be lower, unchanged or higher than the original grade received. If you are still not satisfied with the grades received after the re-grading, escalate your case to the lab instructor to request a review and the lab instructor will finalize the case.

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

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. Since the lab is being done for the second time, we expect that the student will improve upon the older solution. 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, and at 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 the 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 by the offender within twenty-four hours, then a lab grade of zero will automatically be assigned to the offender.

Seeking Help For Labs

- **The C Programming Language.** In this course, we assume you already know how to program in C. If you dont, check out these tutorials:

  https://www.learn-c.org
  https://www.programiz.com/c-programming

  In particular, you will need understand some of the following fundamental concepts of C:

  - Pointers and the use of operators
  - Structs and typedefs
  - Input and output (particularly the syntax of the printf() function)

- **Discussion Forum.** We recommend that students use the Piazza discussion forum to ask the teaching team questions instead of sending individual emails to the teaching staff. For questions related to the current lab project, our target

response time is one business day [2]. *There is no guarantee on the response time to questions about labs other than the current one*. Feel free to answer questions posted by other students.

- **Online Help.** The labs for the course will be offered entirely online and asynchronously. In place of scheduled lab sessions, the labs will be presented using video tutorials supported by slides and the lab manual. Most lab materials will be provided on Learn. We will also be using Learn for lab submissions.

- **Appointments.** If you need individual assistance from the lab instructor or TAs, please contact us by email to arrange a Zoom session. Keep in mind that we cannot and will not help you write or debug your code.

  To make the appointment efficient and effective, when requesting an appointment, please specify three preferred times and roughly how long the appointment needs to be. On average, an appointment is fifteen minutes per project group. In your email that requests the appointment, please also summarize the main questions to be asked.

  Teaching staff will be able to demonstrate how to use the debugger and provide case-specific debugging tips. Teaching staff will not give a direct solution to a lab assignment. Guidances and hints will be provided to help students to find the solution by themselves.

- **Contact Information:**
  Lab Instructor: Ricardo Rolon (rrolon@uwaterloo.ca)
  TA: Alireza Takami Lotfi (alotfita@uwaterloo.ca)
  TA: Shailja Thakur (s7thakur@uwaterloo.ca)
  TA: Samapti Kundu (s25kundu@uwaterloo.ca)
  TA: Ben Zhang (ctzhang@uwaterloo.ca)

- **Additional Links.** The following links may be useful to you during the development of your lab assignments:

  MobaXterm download:
  https://download.mobatek.net/2022020030522248/MobaXterm_Installer_v20.2.zip

  Linux command-line tutorial:
  https://ubuntu.com/tutorials/command-line-for-beginners#1-overview

  User manual for the ddd debugger:
  https://www.gnu.org/software/ddd/manual/pdf/ddd.pdf

[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 a timely response.

Detailed description of the PNG file format:
http://www.libpng.org/pub/png/spec/1.2/PNG-Structure.html

More information about POSIX threads (pthreads):
https://computing.llnl.gov/tutorials/pthreads/

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

Part II

Lab Projects

Lab 1

System Programming in the Linux Computing Environment

1.1 Introduction

1.1.1 Objectives

The purpose of this lab is to introduce system programming in a Linux development environment. After finishing this lab, you will be able to

- use basic Linux commands to interact with the system through the shell
- use standard Linux C programming tools for system programming
- create a program to interact with the Linux file system using the relevant system and library calls.

1.1.2 Topics

Concretely, the lab will cover the following topics:

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

1.2 Starter Files

The starter files are on GitHub at http://github.com/broehl/ece252/ in the lab1/starter directory, which contains the following sub-directories where we have example code and image files to help you get started:

- `cmd_arg` demonstrates how to capture command line input arguments
- `ls` demonstrates how to list all files under a directory and obtain file types
- `png_util` provides a set of utility functions to process a PNG image file
- `pointer` demonstrates how to use pointers to access a C structure
- `images` contains some image files
- `segfault` contains a broken program that has a segmentation fault bug, which you should try to debug during your lab preparation

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

1.3 Preparation

Read the Introduction to the ECE Linux Programming Environment supplementary material in Part III of this lab manual.

You will find the starter files in the png_util directory are helpful. To make your code reusable in the final lab, you should create two functions. One is `is_png()`, which takes eight bytes and checks whether they match the PNG image file signature. The other 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 can be re-used in your lab1 final solution and future lab 2 and lab 3 solutions[1]. 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 code provides the files crc.c and crc.h along with a main.c file that demonstrates how to call the crc function to do the computation.

1.4 Lab Assignment

You will create three programs for this lab – pnginfo, findpng and catpng.

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

1.4.1 The pnginfo command

You will create a command line program named `pnginfo` that prints the dimensions of a valid PNG image file or prints an error message if the input file is not a PNG file or is a corrupted PNG file. The command takes one input argument, which is the pathname of a file. Both absolute pathnames and relative path names are accepted. For example, if the input file is not a PNG file, the command **./pnginfo WEEF_1.png** will output the following lines:

```
WEEF_1.png: 450 x 229
Disguise.png: Not a PNG file
```

If the input file is a PNG file, but some chunks have CRC errors (that is, the CRC value in the chunk does not match the CRC value computed by your program), then the program outputs something similar to what pngcheck does. For example, the 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
```

1.4.2 The findpng command

Next, you will create a tool named `findpng` to search a given directory hierarchy to find all the real PNG files under it. The expected behaviour of the `findpng` command is given in the following "manual page".

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 print 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 relative pathnames[2] of the PNG files, one 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 files under the current working directory. A non-empty search result 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
```

Searching for PNG files under a given directory

The Linux file system is organized as a tree. Every file has a type. Three file types that this assignment will deal with are regular files, directories and symbolic links. A PNG file is a regular file. A directory is (unsurprisingly) a directory file. A link created by `ln -s` is a symbolic link. Read the description of the `stat` family of system calls in section 2 of the Linux man pages for information about other file types. The file `ls/ls_ftype.c` in the starter code gives a sample program to determine the type of a given file. Note that the `struct dirent` returned by the `readdir()` function 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 our departmental Linux servers (eceubuntu) to test your submission. If you want to use the `d_type` field in your code, make sure you test its behaviour on the eceubuntu machines.

[2] i.e. relative to the directory pathname on the command line

To search for all the files under a given directory and its subdirectories, you need to traverse the given directory tree to its leaf nodes. The library call `opendir` returns a directory stream for `readdir` to read each entry in a directory. You need to call `closedir` to close the directory stream once operations on it are 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 file `ls/ls_fname.c` in the starter code contains a sample program that lists all the file entries of a given directory.

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

1.4.3 The catpng command

You are given a directory containing a number of PNG files. Each PNG file is a horizontal strip from a larger imae. All the PNG files 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, ...`. You will concatenate these horizontal image strips sequentially based on the sequence number in the file name to restore the original image. The sequence number indicates the position that the image should be placed in, from top to bottom. For example, the 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 the file `img_1.png` should be followed immediately by the pixel data in the file `img_2.png`. Figure 1.1 illustrates the concatenation order.

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

Man page of catpng

NAME

catpng - concatenate PNG images vertically to create 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.

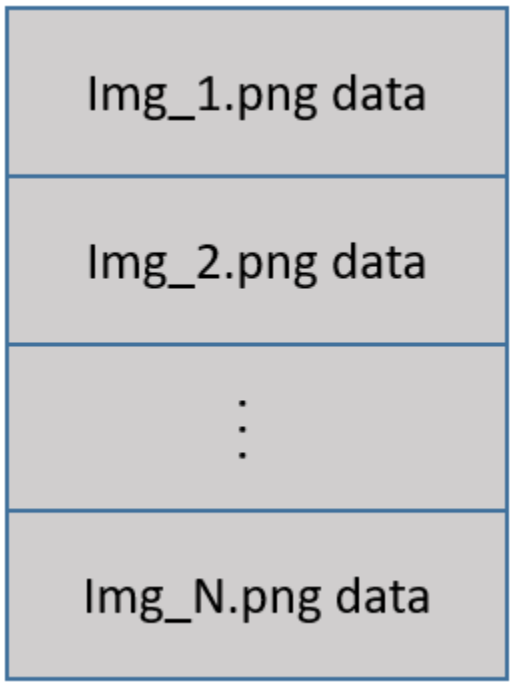


Figure 1.1: Image Concatenation Illustration

File I/O

There are two sets of functions for file I/O operations under Linux. At the system call level, we have the *unbufferred I/O* functions: `open`, `read`, `write`, `lseek` and `close`. At the library call level, we have the 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` calls, so using theses library routines is recommended for this lab.

The `fopen` function returns a FILE pointer given a file name and 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 some number of bytes from the stream pointed to by the FILE pointer returned by `fopen`. Each opened file has an internal state that includes a file position indicator. The file position indicator is set to the beginning of the file when it is opened. The `fread` function will advance the file position indicator by the number of bytes that have been read from the file. The `fseek` function sets the file position indicator to the user-specified location. The `fwrite` function writes a user-specified number of bytes to the stream pointed to by the FILE pointer. The file position indicator also advances by the number of bytes that have 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 are the main reference for details, including function prototypes and how to use them.

PNG File Format

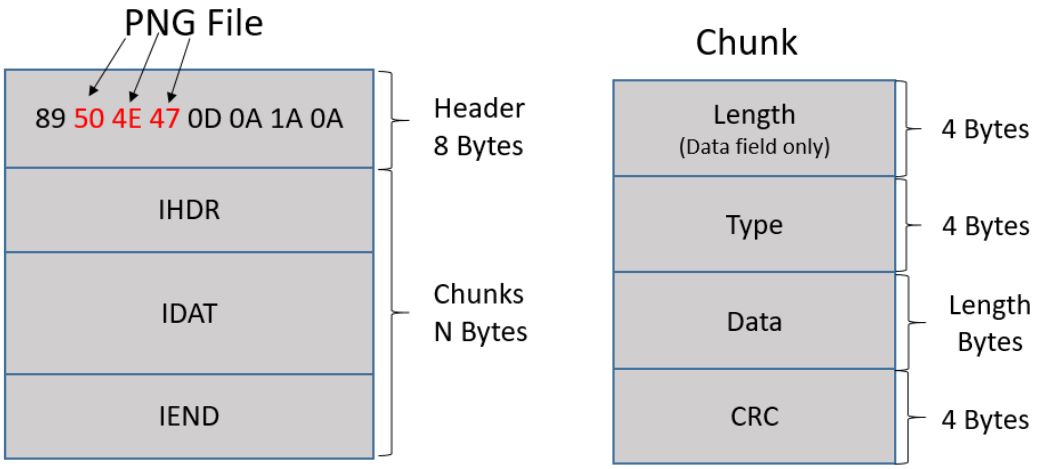
In order to do this assignment, you need to have some understanding of the png file format and how an image is represented in the file. One way to store an image is to use an array of coloured dots referred to as *pixels*. A row of pixels within an image is called a *scanline*. Pixels are ordered from 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 [3] 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 are described in terms of horizontal and vertical pixels.

PNG stands for "Portable Network Graphics". PNG is a 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 codes for 'P', 'N' and 'G' respectively (see Figure 1.2(a)).

The first chunk is the IHDR chunk, which contains meta information about the image such as the dimensions in pixels. The last chunk is always the IEND chunk, which marks the end of the image data stream. In between there is at least one IDAT chunk which contains the compressed, filtered pixel array of the image. There are other types of chunks that may appear between the IHDR chunk and the IEND chunk. For all the PNG files we are dealing with in this assignment, there will only be 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,

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



(a) PNG File Format

(b) PNG Chunk Format

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 also 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 of four ASCII characters. IHDR, IDAT and IEND are the three chunk type codes that this assignment deals with.

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

The CRC field stores a cyclic redundancy check on the preceding 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 in 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 which appear in the order 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

Name	Length	Value
Width	4 bytes	(varies)
Height	4 bytes	(varies)
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

(i.e. RGBA image). The image pixel array data is 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. The Value column in table 1.1 gives the typical IHDR values for the PNG images that you will be processing.

The IDAT chunk data field contains compressed filtered pixel data. For each scanline, an extra byte is added at the very beginning of the pixel array to indicate the filter method used. Filtering is for preparing for the next step of compression. For example, if the raw pixel scanline is 16 bytes long (four pixels), then the scanline after applying the filter will be 17 bytes long. This additional one byte per scanline will help to achieve better compression results. After all scanlines have been filtered, the data is compressed according to the compression method encoded in the 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 approach would be to start with the pixel array for each image and then concatenate the two pixel arrays vertically. Then we would apply the filter to each scanline. Lastly we would compress the filtered pixel array to fill the data field of the new IDAT chunk of the concatenated image.

However, a simpler method 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 an IDAT chunk is compressed using zlib format 1.0. We can use zlib library functions to inflate (i.e. uncompress) the data. The `mem_inf` function in the starter code takes in-memory deflated (i.e. compressed) data as input and stores the uncompressed data in a given memory location. For each IDAT chunk you want to concatenate, you call this function and stack the returned data in the order you wish in order to obtain 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 zlib to deflate (i.e. compress) the in-memory data and returns the deflated (i.e. compressed) data. The `png_util` directory in the starter code demonstrates how to use these two functions.

To create a new PNG file for the concatenated images, an IHDR chunk also needs to have the dimensions of the new PNG file. The rest of the fields of the IHDR chunk can be kept the same as any 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, and the rest of the fields are the same as the input images.

1.5 Deliverables

The following are the steps to create your lab submission.

- Create a directory under your ECE252 directory and name it Lab1.
- Put the entire source code with a Makefile under the directory Lab1. The Makefile default target includes `pnginfo`, `catpng` and `findpng`. That is, the command `make` should generate these three executable files. We also expect that the command `make clean` will remove the object code and all the executables. That is, the `.o` files and the three executable files should be removed.
- Use the `zip` command to zip up the contents of the Lab1 directory and name it lab1.zip. We expect `unzip lab1.zip` will produce a `Lab1` sub-directory in the current working directory and under the `Lab1` sub-directory will be your source code and the Makefile.

Submit the `lab1.zip` file to **Lab 1 Dropbox** in Learn.

1.6 Marking Rubric

Points	Description
5	`Makefile` correctly builds and cleans
20	Implementation of `pnginfo`
30	Implementation of `findpng`
45	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

The purpose of this lab is to learn about and gain practical experience with multi-threaded programming in a Linux environment. A single-thread implementation using blocking I/O to request a resource across the network is provided. You 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, you will have a good understanding of

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

2.2 Starter Files

The starter files are on GitHub at http://github.com/broehl/ece252 in the lab2/starter directory. That directory contains the following sub-directories where we have example code to help you get started:

- `cURL` demonstrates how to use cURL to fetch an image segment from the lab web server
- `fn_ptr` demonstrates how C function pointers work
- `getopt` demonstrates how to parse command line options
- `pthreads` demonstrates how to create two threads
- `times` provides helper functions to measure program execution times

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

2.3 Preparation

Read the entire lab2 manual to understand what the lab assignment is about. Build the starter code and run the executables. Read through the code samples and understand what they do and how they work. Create a single-threaded implementation of the `paster` command (see 2.4.3).

2.4 Lab Assignment

2.4.1 Problem Statement

In the previous lab, a set of horizontal strips of a PNG image file were stored in local files and you were asked to restore the image from those strips. In this lab, the horizontal image segments are on the web. There are 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 incrementing by 1 [2]. Then the web server sleeps for a random time and then sends out a random strip in the simple PNG format that we assumed in lab1. That is, each 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 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's sending you. The HTTP response header has the format "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`
- `ece252-3.uwaterloo.ca`

For example, when you request data from the following URL:
`http://ece252-1.uwaterloo.ca:2520/image?img=1`
you will receive a random horizontal strip of picture 1. If, for example, this random

[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.

strip that 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 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. The server sends a random strip each time. 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 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

You will create an executable called `paster`. The behaviour of the command `paster` is given in the "man page" later in this lab manual.

You should start by creating a single-threaded implementation of `paster` that requests all the image segments from a web server by using blocking I/O and then concatenate these segments together to form the complete image. A single-threaded implementation will be relatively slow, but it will let you sort out all the challenges of accessing files over the web, assembling them and so on before introducing the use of multiple threads. If you are unable to implement threading, you will still get partial marks for a single-threaded implementation (see the marking rubric later in this chapter).

The provided starter code `main_write_header_cb.c` in the `cURL` directory will be helpful. It illustrates how to use the `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. Since you get a random strip each time, the amount of time to get all fifty distinct strips of a picture varies.

You will notice the blocking I/O operation is the main cause of the latency. Your program will be blocked each time it calls `curl_easy_perform()`.

Once you have the single-threaded version working, you will implement a multi-threaded version that reduces the latency of the web operations by sending out multiple blocking I/O requests simultaneously (to different machines) by using `pthreads`. [3].

Your program should create as many threads as the user specifies on the command line, and distribute the work among the three provided servers. Make sure all of your library calls are *thread-safe* (for glibc, e.g. `man 3 printf` to look at the

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

documentation).

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.

When using the pthreads and curl libraries, you need to specify -pthread and -lcurl

The image fragments should be retrieved from:
http://ece252-1.uwaterloo.ca:2520/image?img=n where n is the image number. Keep in mind the image fragments are selected at random, so you may receive the same image fragment multiple times.

2.4.3 Man page of paster

NAME

paster - paste 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 the horizontal image segments of picture 1 from http://ece252-1.uwaterloo.ca:2520/image?img=1 and pastes all distinct segments received from top to bottom in the order of the image segment sequence number. It will output the pasted image to a file named ”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

specifies which image should be retrieved from the web server. Valid values are 1, 2 and 3. Default value is set to 1.

OUTPUT FORMAT

The concatenated image is output to a PNG file with the name ”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 a file named "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

Although sample code for using `libcurl` is provided, you should familiarize yourself with the libcurl API to help you understand how the provided code works. The libcurl documentation can be found at https://curl.haxx.se/libcurl. The man page for each function in the libcurl API can be found at URL https://curl.haxx.se/libcurl/c/allfuncs.html.

Note that the provided example `cURL` code downloads the received image segment to memory and then outputs 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 received segment to a file. An efficient way (i.e. without unnecessary file I/O) is to directly store the received image segment data in memory instead of outputting the data to a file first and then reading the data back from that 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. Here are a few key points from the libcurl manual that are relevant to this lab:

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

2.5.2 The pthreads API

You should read the `pthreads(7)` man page to get an overview of POSIX threads. 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 on 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

Create a multi-threaded implementation of the `paster` command. The following are the steps to create your lab submission.

- Create a directory (under your ECE252 directory) and name it Lab2.
- Put the entire source code with a Makefile under the directory Lab2. The Makefile default target is `paster`. That is, the command `make` should generate the `paster` 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.
- Use the `zip` command to zip up the contents of the Lab2 directory and name it lab2.zip. We expect `unzip lab2.zip` will produce a `Lab2` sub-directory in the current working directory and under the `Lab2` sub-directory will be your source code and the Makefile.

Submit the `lab2.zip` file to **Lab 2 Dropbox** in Learn.

2.7 Marking Rubric

Points	Description
5	`Makefile` correctly builds and cleans
30	Implementation of single-threaded `paster`
65	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

The purpose of this lab is to learn about, and gain practical experience with, interprocess communication and concurrency control in a Linux environment. Shared memory allows multiple processes to share a given region of memory, and it's the fastest way for different processes to communicate. Processes need to take care of concurrency when accessing shared memory, using concurrency control facilities such as mutexes and semaphores that are provided by the operating system.

After this lab, you will be able to

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

3.2 Starter Files

As with the previous labs, the starter files should be downloaded from github. The lab3/starter directory contains the following sub-directories where we have example code to help you get started:

- `fork` has sample code for creating multiple processes and measuring the total execution time; it also demonstrate how a zombie process is created when the parent process does not call `wait` family calls
- `sem` has sample code for using POSIX semaphores shared between processes
- `shm` has sample code for using System V shared memory
- `cURL\_IPC` has sample code for using a shared memory region as a cURL callback 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
- `tools` has a shell script to compute statistics from the timing data and a shell script to clean up IPC resources

The file lab3_eceubunt1.csv is the template that you will need for submitting timing results.

3.3 Lab Preparation

Read the entire lab3 manual to understand what the lab assignment is about. Build and run the starter code to see what it does. You should work through the provided starter code to understand how it works. 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 the `wait(2)` family of system calls.
3. Execute `man ps` to read the man page of the `ps` command.
4. Execute `man shm_overview` to read the Linux man page about POSIX shared memory. At the bottom of the man page, it talks about System V shared memory facilities. Read the corresponding man pages for the System V shared memory API.
5. Execute `man sem_overview` to read the Linux man page about the POSIX semaphore API.
6. Execute `man ipcs` and `man ipcrm` to read the man pages of the Linux IPC facility commands. You will find the `-s` and the `-m` options are helpful in this lab.

Linux man pages are also available online 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 is example code at the end of the man pages.

3.4 Lab Assignment

3.4.1 The Producer Consumer Problem

"Producer-Consumer" 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 in order 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 should the buffer be? Is it true that the bigger the buffer 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 to the one 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. create data) and let consumers extract the image pixel data information (i.e. process data). One easy solution to lab 2 (also commonly seen) is to have each thread do both the producer and the consumer jobs. This implicitly means 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, so having the same number of producers and consumers is not optimal. In addition, in lab2, we did not restrict the receiving data buffer size. In the real world, resources are limited and the situation that a fixed amount of buffer space is available 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 is a

[1] You probably have already noticed in lab2 that once you reach a certain number of threads, you no longer get any 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 a relatively long time to download the image segment.

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 amount of time 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 sent by the server is still in the simple PNG format (see Figure 1.2(a)). The PNG segment is still an 8-bit RGBA 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's sending you. The HTTP response header has the format "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`
- `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 the 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 input arguments specified on the command line. You should save the concatenated image in a file called `"all.png"` in the current working directory. There are three types of work involved. The first is to download the image segments. The second is to process the downloaded image data and copy the processed data to a global data structure for accumulating the concatenated image. The third is to write the 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

[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.

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 a maximum of B items, where each item is an image segment. Each of the horizontal image strips sent out by the lab servers is less than $10,000$ bytes. Each consumer reads image segments out of the buffer, one at a time, sleeping for X milliseconds before each one as specified by the user on 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 its proper place in 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, so 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].

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. Your program is called with the following syntax:

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

The command will execute per the above description and will then print out the

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

[5] Due to network transmission randomness, the order of image segments placed in the buffer may not necessarily be in the same order that they are requested by the producers. The last image segment placed 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 store 50 image segments.

time it took to execute. You should record 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` function for time measurement (see the starter code under the `fork` directory). The output of your program should look like this:

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

For a set of given `(B, P, C, X, N)` values, run your application and measure the time it takes. Note that for a given set of values for `(B, P, C, X, N)`, you need to run your code multiple times to compute the average execution time in a Linux environment. Implement each producer and each 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 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 types of work if you see a need to do so. Just keep in mind that having more processes is not cost free, so 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 should use the System V shared memory API. The bounded buffer is a shared data structure such as a circular queue that all processes have access to. Note that shared memory access needs to be taken care of at the application level. POSIX semaphores are to be used for concurrency control.

The image fragments should be retrieved from
`http://ece252-1.uwaterloo.ca:2530/image?img=n&part=m`
where `n` is the image number and `m` is the fragment number.

To clean up shared memory segments and semaphores, use the `clean_ipcs.sh` script in ece252/lab3/starter/tools

You can redirect the output of any Linux program to a file using `>` as follows:

```
./your_program >somefile.txt
```

Note that the time delays are specified in milliseconds. Linux provides a sleep() function whose parameter is in seconds, and a usleep() function whose parameter is in microseconds.

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. The image segments requested are from image 1.

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

For marking purposes, 2, 4 or 6 time decimals is acceptable.

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 may vary. The exact paster2 execution time your program produces will be different than the one shown in the sample run.

3.5 Deliverables

Put the following items under a directory named Lab3:

1. All the source code and a Makefile. The Makefile default target is the `paster2` executable file. That is, the 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 file should be removed.

2. A timing result .csv file named lab3_hostname.csv which contains the timing results of 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 command line argument values and the timing results. The columns of the .csv file from left to right are the values of B, P, C, X, N, 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 command on `eceubuntu1`. Record the average timing measurement data for the (B, P, C, X, N) values shown in Table 3.1. 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 starter code `run_lab3.sh` is legacy code from previous terms. It does not create the csv file, neither calculates the average as described above, but you can adapt it to be reused. The average should add the resulting time (5 times the same input parameters) and the end result to be divided by 5.

Use the `zip` command to archive and compress the contents of the Lab3 directory and name it `lab3.zip`.

We expect the command `unzip lab3.zip` will produce a `Lab3` sub-directory in the current working directory and under the `Lab3` sub-directory we will find your source code, the Makefile and the lab3_hostname.csv file.

Submit the `lab3.zip` file to **Lab 3 Dropbox** in Learn.

3.6 Marking Rubric

The Rubric for marking is listed in Table 3.2.

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.

Points	Description
5	`Makefile`
20	A partial implementation of `paster2`. The code works correctly with input of (B, P, C, X, N), where $B \geq 1$, $P = 1$, $C = 1$, $X \geq 0$, and $N = 1, 2, 3$.
75	Complete implementation of `paster2` and timing data

Table 3.2: Lab3 Marking Rubric

Lab 4

A Multi-threaded Web Crawler

4.1 Objectives

The purpose of this lab is to design and implement a multi-threaded web crawler. In the previous lab, we practiced memory sharing as a means to communicate between processes. Sharing memory between threads is 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, you will be able to

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

4.2 Starter Files

The starter files are on GitHub, in the lab4/starter directory. That directory contains the following sub-directories where we have example code to help you get started:

- `curl_xml` has example code to show how to use curl and libxml2 together to identify a possible png page and extract http and https links from an html page
- `tools` has a shell script to compute statistics from timing data.

The file lab4_eceubunt1.csv is the template that you will need for submitting timing results.

4.3 Preparation

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

1. Run the given starter code with the following URLs and examine the 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 on condition variables.

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

Create a single-threaded implementation of the `findpng2` command (see Section 4.4.2).

4.4 Lab Assignment

4.4.1 Problem Statement

In the previous labs, the URLs to download the image segments were provided. 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[1].

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

4.4.2 The findpng2 command

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

[1]This lab does not require you 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.

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 store the search results in 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 function to download the data and then processes the downloaded data. The total number of pthread_create() invocations should be 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 unique PNG URLs on the web. It is possible that the number of search results is less than NUM. When this option is not specified, assumes NUM=50.

-v=LOGFILE

log all the URLs visited 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 result 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
```

We'll use the seed: `http://ece252-1.uwaterloo.ca/lab4` or any path forward from it, for testing your lab 4 submission.

4.4.3 Web crawling

The `findpng2` command 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[2]. 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 again.

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 URLs or it reaches the maximum number of PNG URLs specified on the command line.

[2]A valid PNG image is a file whose first 8 bytes match the PNG signature bytes

4.4.4 HTTP

HTTP stands for "Hypertext Transfer Protocol". It carries 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 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 if the link has been relocated. By feeding the `curl_easy_setopt` function the `CURLOPT_FOLLOWLOCATION` option, curl will automatically follow the relocated links. The `CURLOPT_MAXREDIRS` in the curl option setting allows you to specify the 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 that this link has been visited.
- HTTP/1.1 5XX. This is also a broken link, usually caused by a server internal error. We are not able to process the link. But we 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 this lab, we are only interested in two varieties of Content-Type. One is `text/html`, which is an HTML file where we might find more URLs. The other is `image/png`, which is a PNG image. You will process both of those content types, and ignore any others you might encounter.

The http header callback function of curl allows us to process all the header responses from the server. Another way is to use the `curl_easy_getinfo` function to obtain a specific header[3]. For example, with the second parameter of that function set to `CURLINFO_CONTENT_TYPE`, we can obtain the content type header information.

As you may recall from previous labs, the lab server uses an HTTP response header that has the format of "X-Ece252-Fragment: M" where $M \in [0, 49]$ to indicate which image segment it's sending 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.

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

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

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 list of visited URLs is needed. Hashing will make the search very efficient and you might consider using a hash table to represent this already-visited list. The glibc library has a hash table API (`man hsearch(3)`).

4.5 Deliverables

Put the following items under a directory named Lab4:

1. All the source code and a Makefile. The Makefile default target is the `findpng2` executable file. That is, the 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` command 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 command line argument values and the timing results. The columns of the .csv file from left to right are the 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 purposes.

   Run your `findpng2` command on eceubuntu1. Record the average timing 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.

Use the `zip` command to archive and compress the contents of the Lab4 directory and name it `lab4.zip`. We expect that the command `unzip lab4.zip` will produce a `Lab4` sub-directory in the current working directory and under the `Lab4` sub-directory we will find your source code, the Makefile and the lab4_hostname.csv file.

Submit the `lab4.zip` file to **Lab 4 Dropbox** in Learn.

4.6 Marking Rubric

As with the earlier labs, you should start with a single-threaded implementation. This will let you get the basics working, without the added complexity (and increased performance) of using multiple threads. You will receive partial marks for this portion.

Table 4.2 shows the rubric for marking the lab.

Points	Description
5	`Makefile` correctly builds and cleans
20	Implementation of single-threaded `findpng2`
75	Implementation of multi-threaded `findpng2`

Table 4.2: Lab4 Marking Rubric

Lab 5

Asynchronous I/O with cURL

5.1 Objectives

The purpose of 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 making the program concurrent is to use non-blocking, asynchronous I/O. The cURL multi interface enables multiple simultaneous transfers in the same thread.

After this lab, you will be able to

- design and implement a single-threaded concurrent program by using asynchronous I/O
- gain experience with the cURL multi-interface

5.2 Starter Files

As with all the previous labs, the starter files are on GitHub. The lab5/starter directory contains the following sub-directories where we have example code to help you get started:

- `curl_multi` has example code to show how to use the curl multi interface API
- `tools` has a shell script to compute statistics from timing data

The file lab5_eceubunt1.csv is the template that you will need for submitting timing results (see Section 5.5).

5.3 Preparation

Build and run the starter code to see what it does. You should work through the provided starter code to understand how it works. Read the documentation on the curl multi interface at the following URLs:

- https://curl.haxx.se/libcurl/c/libcurl-multi.html
- https://ec.haxx.se/libcurl-drive-multi.html

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. Instead, it should keep multiple concurrent connections to servers open. We will create a single-threaded web crawler named `findpng3` to search the web starting from a given seed URL and find all the URLs that link to PNG images.

5.4.2 The findpng3 command

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

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 write 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 a maximum of 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 unique PNG URLs on the web. It is possible that the number of search results is less than NUM. When this option is not specified, assumes NUM=50.

-v=LOGFILE

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

OUTPUT FORMAT

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

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

The search result 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.

We'll use the seed: `http://ece252-1.uwaterloo.ca/lab4` or any path forward from it, for testing your lab 5 submission.

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 a maximum of 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 might look like the following:

[1]There are two implementation options when you launch up to NUM transfer requests. One is to launch up to NUM transfer requests in a batch, wait for all of them to complete, and then move to the next group of requests. The other is immediate replacement of the individual handle. We recommend the second approach, but either one is fine.

```
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
```

5.5 Deliverables

Put the following items under a directory named Lab5:

1. All the source code and a Makefile. The Makefile default target is the `findpng3` executable file. That is, the 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 command line argument values and the timing results. The columns of the .csv file from left to right are the 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 command 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 $n = 5$.

Use the `zip` command to archive and compress the contents of the Lab5 directory and name it `lab5.zip`. We expect that the command `unzip lab5.zip` will produce a `Lab5` sub-directory in the current working directory and under the `Lab5` sub-directory we will find your source code, the Makefile and the lab5_hostname.csv file.

Submit the `lab5.zip` file to **Lab 5 Dropbox** in Learn.

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.

5.6 Marking Rubric

The Rubric for marking is listed in Table 5.2.

Points	Description
10	Makefile correctly builds and cleans `findpng3`
65	Implementation of `findpng3` in Section 5.4.2
25	Correct timing results in lab5_hostname.csv file

Table 5.2: Lab5 Marking Rubric

Part III

Software Development Environment Reference Guide

Chapter 1

Introduction to ECE Linux Programming Environment

1.1 ECE Linux Servers

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

To access these machines from off campus, you can use the campus VPN. Another way is to first connect to `eceterm.uwaterloo.ca` and then connect to other Linux servers from there using the ssh command. Note that `eceterm` should not be used for computing jobs; it is for accessing other Linux servers on campus. Many of the tools you will be using are not supported on eceterm.

1.2 Basic Software Development Tools

To develop a program, there are three important steps. First, a program is created from source code written by a programmer. Second, the source code is compiled into object code, which is a binary file (on Linux, these files have a ".o" extension). A non-trivial project normally contains more than one source file. Each source file is compiled into one object code file and the linker finally links all the object code to generate the final target, which is the executable that you run. 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 (this is the third step). The debugger is a tool to help you identify and fix these bugs. Table C1 shows the key steps in programming work flow and example tools provided by a general purpose Linux operating system.

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

Most of you are probably more familiar with an Integrated Development Environment (IDE) which integrates all these tools into a single application. Examples of common IDEs are Eclipse and Visual Studio. A different approach is to select a tool for each development 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.

1.2.1 Editor

Some editors are better designed to 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-style editors called *pico* and *nano* are also available for simple editing jobs. 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.

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

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

Listing 1.1: HelloWorld C source Code

1.2.2 C Compiler

The source code then gets fed into a compiler/linker to become an executable program. The GNU project C and C++ compiler and linker 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 give the executable another name instead of the default `a.out`. The `-o` option in gcc allows you to give 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`.

1.2.3 Debugger

The GNU debugger gdb is a command line debugger. Many GUI (i.e. visual) debuggers use gdb as their back-end engine. One popular 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 you 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 helloworld.out

```
gdb helloworld.out
```

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

Compared to the 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 the gdb console in the bottom window with the source code window above it. You can see the values of variables in the data window, which is above the source code window. Select View to toggle all of these windows.

A good gdb/ddd tutorial can be found at https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_gdb.php. Pay special attention to the section on how to troubleshoot segmentation fault problems.

1.3 More on Development Tools

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

1.3.1 How to Automate Builds

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

In a Makefile, you specify the targets to be built, what prerequisites the target depends on, and what commands are used to build the target. These are the *rules* contained in a Makefile. Makefiles have their 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 the 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. The first is to *compile* the source code into an object code .o file. The 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 the compilation and linking stages gives 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 `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 have also added a target named *clean* so that `make clean` will clean the build.

So far we have seen that the Makefile contains *explicit rules*. A 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 a .o file for each .c file. See http://www.gnu.org/software/make/manual/make.html to explore more about makefiles.

1.3.2 Version Control Software

We use the Git version control software. 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.

1.3.3 Integrated Development Environment

Eclipse with the C/C++ Plug-in has been installed on all ECE Linux servers. Type the following command to bring up the Eclipse frontend.

```
/opt/eclipse64/eclipse
```

This Eclipse is not the same as the default Eclipse under the `/usr/bin` directory. You may find running Eclipse over a network performs poorly at home though. It depends on how fast your network speed is.

If you have a Linux operating system installed on your own personal computer, then you can download Eclipse with the C/C++ plugin from the Eclipse web site and then run it on your own local computer. However you should always make sure that your program will also work on the eceubuntu machines, since that is the environment that the TAs will be using to test your code.

1.4 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.

Manual 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 from section 1 of the manual.

You can also use `man -k` or `apropos` followed by a string to obtain a list of man pages that contain that string, using the Whatis database. You can run `man whatis` to see more details about `whatis`.

Appendix A

Forms

Lab administration related forms are given in this appendix.

ECE252 Request to Leave a 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.