Final Exam: CS 35L Section 1

University of California, Los Angeles

Spring 2018

Name: _____

Student ID: _____

Question	Points
1	**/12**
2	**/20**
3	**/16**
4	**/16**
5	**/18**
6	**/18**
Total	

Instructions:

1. This examination contains 16 pages in total, including this page.
2. You have **three (3) hours** to complete the examination.
3. Write your answers in this exam paper. The draft papers will not be accepted. If you running out of space, please let me know.
4. You may use any printed resources, including lecture notes, books, slides, assignments. **You cannot use any electronics.**
5. Please finish the final **independently**. If you have any question, please raise your hand.
6. There will be partial credit for each question.
7. Best of luck!

Question 1: Concept Question

[12 pts] Choose **two (2)** of the following three questions and answer them concisely. **DO NOT** answer all of them.

(a) What are dynamic linking and static linking? List two benefits for each one. Why we need linking for C in the first place?

(b) Suppose the remote server has created a pair of username and password for you. Briefly describe the workflow to enable password-less login. You do not need to write the command; just explain each step. Suppose you start from no public-private key pair on your own computer.

(c) Assume you want to cheat for CS 35L assignments. You see a Github repository which is exactly what you need. What is the command if you want to save a local copy of it? If the author of the repo makes a few changes and you want to sync up locally, what is the command for this purpose? What is the command if the author wants to upload some local commits? What is the benefit of Git compared to the old SVN?

Question 2: Unix Command

[20 pts]

The following is the subset of the man page for the "top" command.

```
TOP(1)                          User Commands                          TOP(1)

NAME         top
       top - display Linux processes
SYNOPSIS         top
       top -hv|-bcEHiOSs1 -d secs -n max -u|U user -p pid -o fld -w [cols]

       The traditional switches '-' and whitespace are optional.
DESCRIPTION         top
       The top program provides a dynamic real-time view of a running
       system.  It can display system summary information as well as a list
       of processes or threads currently being managed by the Linux kernel.
       The types of system summary information shown and the types, order
       and size of information displayed for processes are all user
       configurable and that configuration can be made persistent across
       restarts.

       The program provides a limited interactive interface for process
       manipulation as well as a much more extensive interface for personal
       configuration  --  encompassing every aspect of its operation.  And
       while top is referred to throughout this document, you are free to
       name the program anything you wish.  That new name, possibly an
       alias, will then be reflected on top's display and used when reading
       and writing a configuration file.


The command-line syntax for top consists of:

         -hv|-bcEHiOSs1 -d secs -n max -u|U user -p pid -o fld -w [cols]

       The typically mandatory switch ('-') and even whitespace are
       completely optional.

       -b  :Batch-mode operation
            Starts top in Batch mode, which could be useful for sending
            output from top to other programs or to a file.  In this mode,
            top will not accept input and runs until the iterations limit
            you've set with the '-n' command-line option or until killed.

       -c  :Command-line/Program-name toggle
            Starts top with the last remembered 'c' state reversed.  Thus,
            if top was displaying command lines, now that field will show
            program names, and vice versa.  See the 'c' interactive command
            for additional information.

       -d  :Delay-time interval as:  -d ss.t (secs.tenths)
            Specifies the delay between screen updates, and overrides the
            corresponding value in one's personal configuration file or the
            startup default.  Later this can be changed with the 'd' or 's'
```

interactive commands.

Fractional seconds are honored, but a negative number is not allowed. In all cases, however, such changes are prohibited if top is running in Secure mode, except for root (unless the 's' command-line option was used). For additional information on Secure mode see topic 6d. SYSTEM Restrictions File.

-E :Extend-Memory-Scaling as: -E k | m | g | t | p | e
Instructs top to force summary area memory to be scaled as:
k - kibibytes
m - mebibytes
g - gibibytes
t - tebibytes
p - pebibytes
e - exbibytes

Later this can be changed with the 'E' command toggle.

-n :Number-of-iterations limit as: -n number
Specifies the maximum number of iterations, or frames, top should produce before ending.

-o :Override-sort-field as: -o fieldname
Specifies the name of the field on which tasks will be sorted, independent of what is reflected in the configuration file. You can prepend a '+' or '-' to the field name to also override the sort direction. A leading '+' will force sorting high to low, whereas a '-' will ensure a low to high ordering.

This option exists primarily to support automated/scripted batch mode operation.

-s :Secure-mode operation
Starts top with secure mode forced, even for root. This mode is far better controlled through a system configuration file (see topic 6. FILES).

-S :Cumulative-time toggle
Starts top with the last remembered 'S' state reversed. When Cumulative time mode is On, each process is listed with the cpu time that it and its dead children have used. See the 'S' interactive command for additional information regarding this mode.

-u | -U :User-filter-mode as: -u | -U number or name
Display only processes with a user id or user name matching that given. The '-u' option matches on effective user whereas the '-U' option matches on any user (real, effective, saved, or filesystem).

Prepending an exclamation point ('!') to the user id or name instructs top to display only processes with users not matching

```
the one provided.

The 'p', 'u' and 'U' command-line options are mutually
exclusive.
```

Given the above instructions, you should specify the sequence of commands you use to finish the following tasks:

(a) Write a single command using top to display the Linux processes. The command should refresh the display every 2.5 seconds, and terminate after 10 iterations.

(b) Write a single command using top to display the Linux processes. The command should operate in secure mode. The command should exclude the processes belonging to user "root". Save the results to a file called top.txt under your HOME directory.

(c) Go to your HOME directory. For top.txt, grant r/w/x permission to the owner, r/x permission to the group, and r permission to anyone.

(d) Change the filename to top_2.txt, create a new directory called test under HOME directory, and move the file into this directory.

(e) Prepend this directory (test) to the PATH variable

(f) Use sed command to extract the last path directory in PATH variable. (i.e. if PATH is "/a/b:/a/c:/a/e/f" your command should output "/a/e/f")

Please answer each of them *respectively*.

Question 3: Bash Programming

[16 pts] Write a bash script "trivial" to finish the following tasks:

(a) The script takes two command line arguments. The first argument is a filename. The second one is the path of a directory.

(b) This script then checks whether the filename indicated by the first argument exists in that directory (Yon do not have to handle the extension). If not, create such a file.

(c) Count the number of the files that have a name with more than 3 characters under this directory. Denote this number as x. (Again, do not worry about the extension.)

(d) Count all the symbolic links under this directory. Denote this number as y.

(e) Write a function which calculates $1/x + y^3 + x \cdot y$. Call this function on x and y and print the result to the standard output.

(f) Your script should handle two exceptions: when the input parameter number does not satisfy the requirement, and when the second argument is not a valid directory. Exit the program under these two situations.

Question 4: Python Programming

[16 pts] In this question, you shall write a Python script to test if a positive integer n is a **highly composite number**. Do not be scared away from the question; It is easy after you read and understand the following explanation (at most 15 minutes of reading). Hope that you also learn something from this question.

In number theory, a divisor of an integer n is an integer m that can be divided by n. That being said, 10 has 4 positive divisor (1, 2, 5, 10) and 12 has 6 (1, 2, 3, 4, 6, 12). We will later in this question denote this as $d(n)$ (i.e., $d(10) = 4, d(12) = 6$). Thus, we have $d(1) = 1, d(2) = 2, d(3) = 2, d(4) = 3, d(5) = 2, d(6) = 4$.

A **highly composite number** is a positive integer with more positive divisors than any smaller positive integer has. In other words, n is a highly composite number if and only if $\forall m < n, d(n) > d(m)$ (or, in plain English, $d(n)$ is greater than $d(m)$ for any m, as long as $m < n$).

With the above definition, by checking $d(1)$ to $d(6)$, we can see that 1, 2, 4 and 6 are highly composite number, while 3, 5 are not.

Therefore, you can follow the following steps to check whether an integer n is a highly composite number.

- Calculate $d(n)$;
- Enumerate every integer from 1 to $n - 1$. For each integer m, calculate $d(m)$;
- If $d(n)$ is greater than any $d(m)$, n is a highly composite number; otherwise it is not.

There exists an efficient way to calculate $d(n)$. The critical step is to calculate prime factorization of a given n, which means that we will express an integer as the multiplication of a set of prime integers. For example, $360 = 2 \times 2 \times 2 \times 3 \times 3 \times 5 = 2^3 \times 3^2 \times 5^1$. After we get this factorization, it is easy to calculate the $d(n)$. Suppose that $n = p_1^{k_1} \cdots p_r^{k_r}$, where $p_1 \cdots p_r$ are different prime numbers, $d(n) = (k_1 + 1)(k_2 + 1) \cdots (k_r + 1)$. For example, $10 = 2^1 \times 5^1$, and $d(10) = (1 + 1) \times (1 + 1) = 4$; $360 = 2^3 \times 3^2 \times 5^1$, and $d(360) = (3 + 1) \times (2 + 1) \times (1 + 1) = 24$. To make sure you understand, try calculating $d(100)$ on a draft paper. If you get 9, you are good to continue.

Luckily enough, you are not asked to calculate the prime factorization of n. Instead, there exists an library called zhaowei that helps you do it. In this library, you can call a function called "factorization", which takes an integer as the input, and outputs a dictionary which represents the prime factorization of the input.

Calls factorization(10) returns a dictionary {2:1, 5:1}, since $10 = 2^1 \times 5^1$.
Calls factorization(360) returns a dictionary {2:3, 3:2, 5:1}, since $360 = 2^3 \times 3^2 \times 5^1$.

Note that if the input for factorization is not an integer greater than 1, the output is an empty dictionary. E.g. factorization(0.2) and factorization(-1) will return { }.

(a) Write a Python 2 script (please specify if you use Python 3 instead) that takes one argument n from the command. Check whether n is a highly composite number. You are encouraged to utilize the library zhaowei and the function factorization.

(b) Your script should check the option -f. If -f exists, reads one extra argument which is the filename. You should open the file with this name and save the result (whether n is a highly composite number) "Yes" or "No" in the file. If -f does not exist, output "Yes" or "No" to the standard output.

(c) Your script should contain a function calculate-d, which takes one argument m and returns $d(m)$.

(d) Your script should output "error" to the standard output and terminate if the input is not a positive integer. You **DO NOT** have to consider any other exception (e.g. file does not exist).

Bonus point (5 points, added directly to your total score): Instead of "Yes" or "No", output the list of highly composite number from 1 to n. E.g. `python your_script_name.py 6` should output [1,2,4,6] instead of "Yes". You earn 3 extra points for successfully outputting the list. You earn the rest 2 points if your program is efficient. Write your bonus program separately **on the last page (page 16)**. Simply print to standard output and ignore any error in this case.

Hint: You script is considered inefficient if you keep repeating the general three steps (calculate $\mathrm{d}(n) \to$ loop from 1 to $n-1 \to$ compare).

Question 5: C Programming: Multithreading, Git

[18 pts] You have a local Git project. This Git project only has one branch master. Inside this directory is a single file called matrix.c. Inside the file it reads:

```
#include <stdio.h>
#include <stdlib.h>
#define m_size 16
int mat_a[m_size][m_size];
int mat_b[m_size][m_size];
int mat_c[m_size][m_size];
int main () {
  int i; int j; int k;
  for (i = 0; i < m_size; i += 1) {
    for (j = 0; j < m_size; j += 1) {
      mat_a[i][j] = rand() % 10;
      mat_b[i][j] = rand() % 10;
      mat_c[i][j] = 0;
    }
  }
  for (i = 0; i < m_size; i += 1) {
    for (j = 0; j < m_size; j += 1) {
      for (k = 0; k < m_size; k += 1) {
        mat_c[i][j] += mat_a[i][k] * mat_b[k][j];
      }
    }
  }
  return 0;
}
```

You are asked to do the following.

(a) Create a new branch called "test" and switch to it.

(b) In this new branch, create a new file called mt_matrix.c.

(c) Understand the code in matrix.c; write a multithreading version with **4** threads in mt_matrix.c. You only have to convert the part where we calculate mat_c (not initialization) to multithreading version.

(d) Your program should be robust against errors during multithreading.

(e) Commit the changes and merge to the master branch.

Please write your commands on this page, and write your program on the next page.

Question 6: C Programming: System Call and Compiling

[18 pts]

(a) Suppose that you have a file called input.txt under your HOME directory. Write a main.c script. In the main function use system calls to read the content in this file. Remember to close it afterwards.

(b) Suppose that the content in input.txt is an integer. Save the integer you read to some variable.

(c) Write a C file called repeat.c and its header repeat.h. In these files write a function repeat_random with one argument n. It generates a random string with the length n. It then repeats itself and thus generates a string with length $2n$. The function should return both the random string and its repeated version. E.g. repeat_random(3) possibly returns a data structure with "acz" and "aczacz".

(d) In main.c, call the function repeat_random in repeat.c with the integer you just read from input.txt. You should dynamically load repeat_random function in repeat.c using dlopen and dlsym. You are supposed to close up in the end as well.

(e) Use system call to write the repeated random string to the standard output.

(f) Your program should consider all the linking errors and system call errors. If an error occurs, simply returns -1. Otherwise, a successful main should return 0.

(g) Write a simple makefile that compiles your library and program.

Hint: You can refer to this piece of code on how to generate a random letter.

```
char randomletter = ‘‘abcdefghijklmnopqrstuvwxyz’’ [random () % 26];
```

Reference on system calls:

```
int open(const char *path, O_WRONLY | O_APPEND); // The return value
is the file descriptor for the file indicated in path. Returns negative upon failure.

int close(int fildes); // Returns -1 upon failure.
```

Please write your makefile and repeat.h on this page, and repeat.c, main.c on the next page.