Assignment 5: Asynchronous socket communication

Start Assignment

Due Saturday by 10:59am **Points** 160 **Submitting** a file upload **File Types** zip
Available Aug 1 at 7pm - Aug 14 at 10:59am 13 days

Introduction

This project will require that you compute perfect numbers. These perfect numbers are numbers such that the sum of their proper divisors is equal to the number itself. For instance, the proper divisors of 6 are 1, 2, and 3, which sum to 6.

Learning Outcomes

After successful completion of this assignment, you should be able to do the following

- Compare and contrast IPC facilities for communication (Module 7, MLO 2)
- Explain the Client-Server communication model at a high level (Module 8, MLO 1)
- Understand and use the programmer's view of the internet to design network programs (Module 8, MLO 3)
- Explain the concept of Unix sockets (Module 8, MLO 4)
- Design and implement client and server programs for IPC using sockets (Module 8, MLO 5)
- ompare and evaluate designs for servers (Module 8, MLO 6)

Specifications

You will write 3 related programs to `manage`, `report`, and `compute` results. You are **required** to brute force the algorithm. Yes, this is ridiculous, but the algorithm is NOT the point of this assignment.

`compute`'s job is to compute perfect numbers. It tests all numbers beginning from its starting point, subject to the constraints below. There may be more than one copy of compute running simultaneously.

`manage`'s job is to maintain the results of `compute`. It also keeps track of the active `compute` processes, so that it can signal them to terminate, including the host name and performance characteristics. You will need to use one of `poll` or `select` in order to monitor the different computation processes for results in order to respond correctly to the report process.

`report`'s job is to report on the perfect numbers found, the number tested, and the processes currently computing. If invoked with the `-k` switch, it also is used to inform the `manage` process to shut down computation. `report` will query the server for information, provide that to the caller, then shut down.

Use sockets for communication, with `manage` as the "master" process, or in other words, the server.

In order for the `compute` process to know what values to check, it will query the server for a range of values which it will check for perfect numbers. The server should send back a range that will take approximately 15 seconds for the client to check. In order for the server to know this, the clients must send some sort of information about their performance characteristics. To determine these performance characteristics (think FLOPS/IOPS), you will make use of a timing loop, where you execute a known number of operations, and time how long they take.

All processes should terminate cleanly on `INTR`, `QUIT`, and `HANGUP` signals. When the `-k` is flag is used on `report`, report sends a message to `manage` to trigger an INTR signal, which forces the same shutdown procedure. You will have to implement a standard to use via sockets to tell the other side to send itself the signal, via `kill`.

You will be implementing `manage` and `report` using python, while compute will be done in C. `compute` must utilize a separate thread for monitoring the socket for a shut down message from the server. I would suggest making use of some signal handler to deal with messages from the server.

Hints

Where to Start

et a working python networking server going. I'd suggest starting a simple echo server. The unctions are named the same as their C counterparts, and live in the socket library. Pick a port value that is in the 50,000 range -- if you get an error about the port already in use, move to a new port number and try again.

2. Learn how to use "select" or "poll" in python; add this to your server process to handle the multiplexing over the currently running clients.
3. Write some timing code in C to calculate performance characteristics. You can use something like the below --

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <unistd.h>

static inline uint64_t rdtsc() {
        uint32_t lo, hi;
        __asm__ __volatile__ ("xorl %%eax, %%eax\n cpuid\n rdtsc\n"
                              : "=a" (lo), "=d" (hi)
                              :
                              : "%ebx", "%ecx");
        return (uint64_t)hi << 32 | lo;
}
```

```
#define ITERS 10000

int main(int argc, char **argv)
{
        //get CPU clock speed, assumes the following is running in the background or in anothe
r logged in terminal:
        // (while true ; do cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_cur_freq | R -q -
e "x <- read.csv(file('stdin'), header = F); summary(x); sd(x[ , 1])" | grep Mean | cut -f2 -
d':' >| ~/ticks.txt; sleep 1; done) &
        //make sure to kill that loop when you're done with testing/running your code -- bring
it to the foreground with fg then hit control-C
        //there may be better ways to do this, but that gives the current speed, rather than m
ax speed
        FILE *f = fopen("ticks.txt", "r");
        long ticks;
        fscanf(f, "%ld", &ticks);

        //start timing!
        uint64_t time = rdtsc(), total;
        for(volatile int i = 0; i < ITERS; ++i){
                //do something here to time -- repeat it 10 times or so (actual lines of code)
        }
        //get time for timed operation AND looping
        total = rdtsc() - time;

        time = rdtsc();
        for(volatile int i = 0; i < ITERS; ++i){
                //don't actually do anything here, this measures loop overhead
        }
        //subtract off loop overhead!
        total -= (rdtsc() - time);

        //you won't actually be printing, but this shows your results
        //if numbers seem weird, increase iterations or number of copied lines of code of time
d operation
        printf("%d iterations took %lu cycles\n", ITERS, total);
        printf("average iteration time is %f seconds (clockspeed = %ld Hz)\n", (double)total/(
double)ticks, ticks);

        return 0;
}
```

4. Write a thread which monitors a socket and can send a signal.
5. Write the report process in python.
6. Finish the remainder of the requirements!

Sending and Receiving Data

Recall that when sending data, not all of the data may get written with just one call to `send`. Similarly, when receiving data, not all the data may be read by one call to `recv`. This occurs because of network interruptions, server load, and other factors. You'll need to carefully watch the number of characters read and/or written, as appropriate.

Structured messages are definitely the way to go here. Defining a data exchange format would be immensely useful!

What to turn in?

- You can only use C and python for coding this assignment and you must use the gcc compiler and python 3.
- You must use the C11 standard on os1.
- You must use python 3.8 on os1.
- For gcc, you must use (at least) the flags `-std=c11 -Wall -Werror -g3 -O0` -- this means that all warnings **must** be eliminated in order to compile correctly.
- Your assignment will be graded on os1.
- Submit a single zip file containing the following.
  1. All of your program code, which can be in as many different files as you want
  2. The compilation script named `compileall` or a `Makefile`.
- This zip file must be named `youronid_program5.zip` where youronid must be replaced by your own ONID.
  - E.g., if chaudhrn was submitting the assignment, the file must be named `chaudhrn_program5.zip`.
- When you resubmit a file in Canvas, Canvas can attach a suffix to the file, e.g., the file name may become `chaudhrn_program5-1.zip`. Don't worry about this name change as no points will be deducted because of this.

Grading

- This assignment is worth 15% of your final grade.
- compute is worth 40 points
- anage is worth 40 points
- report is worth 40 points
- Having them all interact properly is worth 40 points